UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

MITIGATING THE NOISY NEIGHBOR PROBLEM: IMPROVING ISOLATION OF CONTAINERS ON MULTI-TENANT INTEL CLOUD SERVERS

JAKOB EBERHARDT

Thesis supervisor
JORDI GUITART FERNANDEZ (Department of Computer Architecture)

Degree
Master's Degree in Innovation and Research in Informatics (High Performance Computing)

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

27/01/2026

Abstract

Running multiple containers on a cloud server simultaneously enables higher utilization and efficiency. However, depending on the workload, it may cause a significant performance impact due to interference on shared resources such as last-level caches and memory bandwidth. This issue is commonly known as the noisy neighbor problem. In this work, we go beyond the typical mitigation of co-locating only compatible workloads on the same server. To this end, we develop and test novel models that improve the isolation of containers based on their potential interference and resource demand. To enhance decision-making and isolation, we employ the Intel Resource Director Technology (RDT) [1], which enables monitoring and allocation of shared resources.

We experimentally demonstrate how a dynamic scheduling and isolation policy exploits the nonlinear resource-to-performance relationship of certain workloads. Typically, some workloads can reach a locality tipping point. Hence, they perform significantly better once they are allocated sufficient exclusive resources. Workloads that are unlikely to ever reach this point are only penalized linearly if their allocation is reduced. We show how this can enable an average efficiency improvement of 14.7% for jobs that can reach sufficient allocations while imposing only a 0.9% penalty on the remaining workloads. This results in an overall efficiency improvement of 4.9%. For specific load scenarios, the efficiency of certain workloads can even be increased by up to 42% while maintaining a penalty of 0.4% to the remaining jobs. We introduce a sensitivity-probing process that leverages resource partitioning to infer workload-specific sensitivities in noisy environments. Additionally, we provide a detailed analysis of the interaction between cache slicing and resource partitioning. We demonstrate how workloads can incur orders of magnitude more cache misses depending on the size of the memory pages backing them.

Acknowledgement

Many thanks to all my professors at UPC. Special thanks to my supervisor, Jordi, for his dedication and constructive attitude. CloudLab US enabled this project by backing us with testbeds. I also thank my friends for making my studies what they were. Thank you, Beatriz, for your support, encouragement, and for the incredible time with you.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Chapter 1

Introduction

Data center operators are under constant pressure to enable higher utilization of available compute resources. By increasingly tapping into the servers' potential, operators can apportion the fixed costs associated with running, maintaining, or cooling the servers, thereby increasing overall efficiency. Since most customers typically only require a fraction [2, 3] of the compute resources available on a common cloud server, the most significant levers in this regard are virtualization and multi-tenancy [4, 5].

By employing virtualization, the underlying resources of a server can be partitioned into appropriate, isolated units that can be adapted to meet customer demand. In multi-tenant environments, these virtual units, such as virtual machines or containers, are typically scheduled on hardware using time- or space-sharing. In the case of virtual machines, full virtualization provides stronger hardware-level isolation, but it incurs the overhead of each guest running its own kernel. Meanwhile, container virtualization relies on process-level isolation, where all guests share a single kernel. Even though this partially loses the isolation, containers are becoming increasingly popular for running workloads and hosting applications. In both cases, there are concerns regarding the security [6, 7] and quality of service (QoS) in multi-tenant environments.

In this work, we will focus on the latter aspect, in particular on the *noisy neighbor* problem. This effect describes a performance degradation [8, 9, 10] of a workload running inside a virtual unit due to the activity caused by other virtual units running on the same physical host. Historically, this effect has been difficult to mitigate because it is subject to design choices set in the microarchitecture of common multi-core processors. Most notably, this includes resources that are typically shared across multiple cores, such as the last-level cache (LLC) and the bandwidth to main memory. Interfaces for network cards, hard drives, or discrete accelerators are also considered shared resources. However, mitigations for interference and contention of these resources can employ established, software-based mechanisms and interfaces. These are provided by the operating system or the respective runtimes, e.g., to monitor guest usage patterns and limit their quotas to guarantee a defined level of quality. In con-

trast, detecting *concrete* interference problems at the microarchitecture level during runtime poses a major challenge.

Isolating the distinct interference potential of a given workload becomes increasingly difficult and impractical as the number of concurrently running workloads on a CPU increases. If the operators know exactly which workloads will be executed, the interference profile of these workloads can be probed ahead of time to possibly improve the scheduling. Yet this so-called offline profiling adds complexity and is infeasible for cloud vendors, which typically do not know which workloads are executed within their customers' virtual compute units. Additionally, this approach does not consider possible changes to a workload's interference profile, including its configuration or the phases it can execute during its lifetime. Even if one could conduct accurate profiles during the live execution, resolving conflicting interference patterns is mostly limited to moving incompatible workloads to separate nodes, which can decrease utilization in favor of quality of service.

1.1 Motivational Example

In this section, we present a concrete example of the noisy neighbor problem. In Figure 1.1, we see the Instructions per Cycle (IPC) executed within a container. It is hosted on a multi-tenant server with one CPU featuring sixteen cores. Specifically, the container is running a database that is under constant high load. The IPC indicates how well the database is running, e.g., how many transactions per second it can process. Initially, the database executes about 0.90 useful instructions per cycle, which is normal for data-intensive workloads. However, the performance drops significantly to 0.77 after 70 seconds.

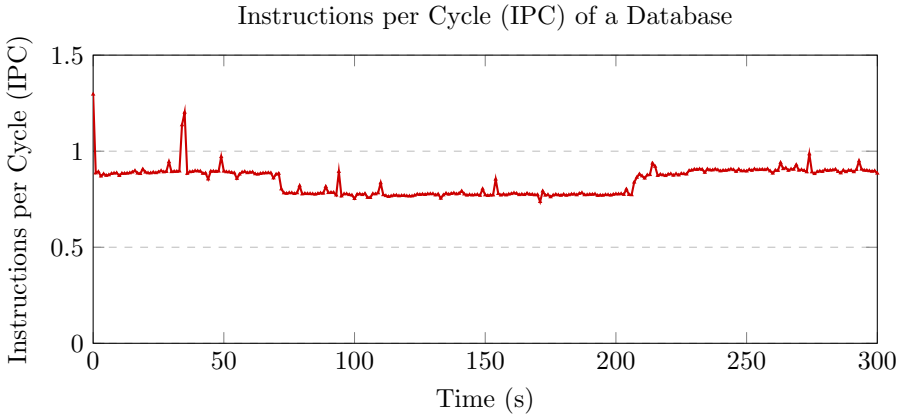
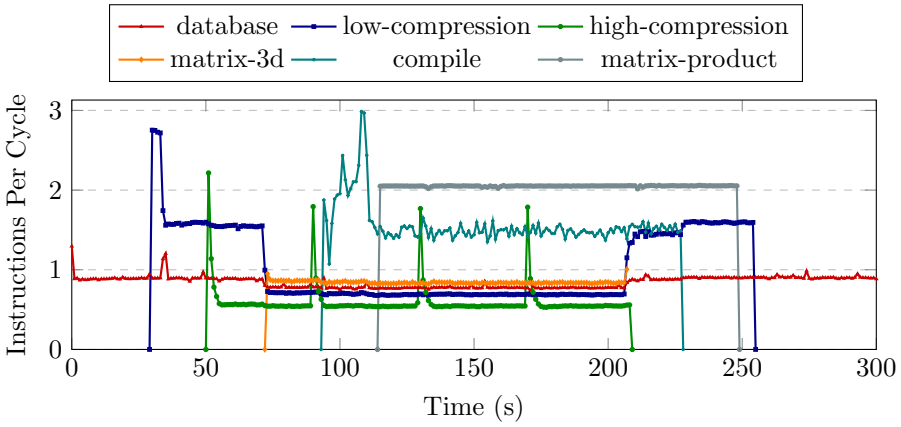


Figure 1.1: The Instructions per Cycle (IPC) executed by a container running a database under steady load.

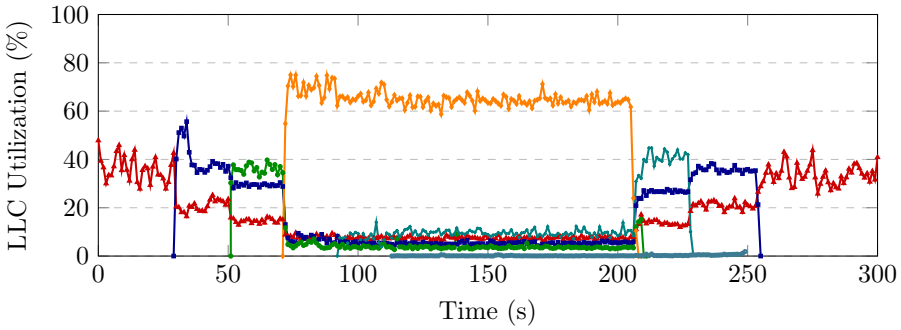
If we consider Figure 1.2a, we see the other container jobs that arrive and are executed on the host. Prior to the sudden IPC drop, the database was already sharing the host with two compression jobs, which did not affect the database. However, once the 3D matrix job arrives after 70 seconds, the database and the low-compression job suffer performance issues while the high-compression job continues its execution relatively stable.

In Figure 1.2b, we can see that the matrix 3D job aggressively starts utilizing the last-level cache and occupies over 60% percent of its total capacity for the full duration of the job. In contrast, we can see that the database previously tolerated the compression jobs, which already used the cache. The matrix *product* job, however, is not data-intensive. Hence, it does not use the last-level cache as much, keeping its IPC stable throughout. Similarly, the compilation job is unaffected by pressure on the last-level cache, even though it appears to fill it, as seen when the 3D matrix container stops. In particular, we can observe that the cache-intensive period led to increased execution stalls, as shown in Figure 1.2c. However, depending on the workload, the increase in stalls and the consequent drop in IPC vary. For example, low-compression jobs spend twice as many of their cycles stalling, while the impact on the database is more subtle. The high compression job follows an iterative pattern. In fact, most of its cycles are spent stalling the execution, independent of its neighbors.

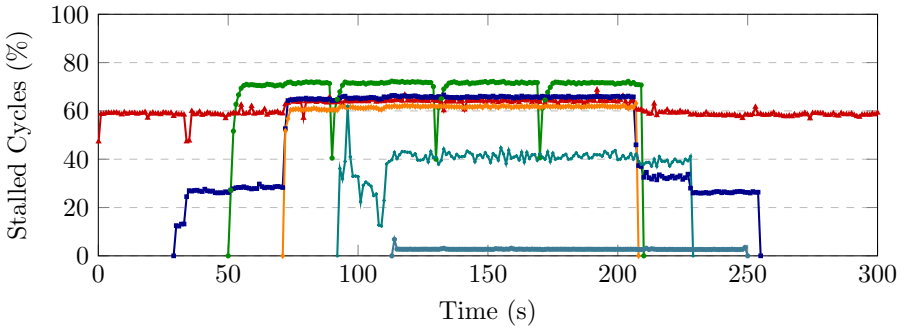
Hence, a given workload can respond differently to specific interference patterns, extending the noisy neighbor problem beyond CPU- or memory-bound applications. Additionally, workloads can have execution phases, such as compilation or high-compression jobs, which require dynamic monitoring and handling.



(a) IPC



(b) LLC Utilization (%)



(c) Stalled Cycles (%)

Figure 1.2: Overview of container metrics, which are executed on a shared host. Each container runs on an exclusive CPU core. However, the 24 MB last-level cache and the memory bandwidth are shared.

1.2 Contributions

In this work, we will address the noisy neighbor problem by employing the Intel Resource Director Technology (RDT), which enables flexible partitioning and allocation of the last-level cache and memory bandwidth. We define and implement a dynamic node-level scheduling policy to mitigate the noisy neighbor problem at the container level. To this end, we use resource partitioning of shared resources provided by the Intel Resource Director Technology to increase the global execution efficiency.

By analyzing the resource-to-performance relationship of different workloads, we can increase the global efficiency by up to 4.9%. To enable this, we develop a fully online probing process that accurately reports a container's resource demand, boosting its performance while imposing negligible penalties on other workloads. Additionally, we demonstrate how resource isolation enables accurate sensitivity probing and quantification without the need to maintain a separate, noise-free host.

We introduce a comprehensive experimental setup that enables reliable and reproducible benchmarks in this domain. Further, it facilitates the implementation of scheduling policies by exposing robust interfaces that integrate all aspects needed to mitigate the noisy neighbor problem. Most importantly, this includes the RDT interface for resource partitioning and the relevant real-time container monitoring data.

Additionally, we analyze the interaction between cache slicing and last-level cache partitioning on Intel hardware. Specifically, we demonstrate how memory page sizes influence conflict cache misses in combination with smaller resource partitions. We demonstrate how large memory pages can mitigate this effect and are therefore generally relevant when using RDT cache partitioning.

1.3 Thesis Structure

This thesis is organized as follows: Chapter 3 introduces the fundamental methodology of this work, which includes the definition of the metrics and optimization goals. In Chapter 4, we present the experimental setup that was developed during this project and used as a foundation for running and analyzing experiments. Additionally, we detail a significant performance limitation that involves the interaction between sliced last-level caches and Intel Resource Director Technology partitioning. Chapter 5 includes the definition of the experiments, which are further validated in Chapter 6. We evaluate the results in Chapter 7. Chapter 8 concerns the impact on sustainability and ethical implications of this work, and in Chapter 9, we draw a conclusion.

Chapter 2

Background

In this chapter, we provide a summary of the relevant background related to the noisy neighbor problem. This includes an overview of the involved memory hierarchy on modern Intel hardware and the shared resource model. We introduce the Intel Resource Director Technology and the functionalities that are used in this project. Further, we summarize the relevant literature related to the noisy neighbor problem and RDT.

2.1 Memory Hierarchy & Shared Resources

In modern hardware, a single silicon die features multiple homogeneous or heterogeneous physical cores. A hierarchy of caches enables higher levels of performance. These include first-level instruction and data caches, as well as a unified level-two cache. Typically, these caches are private, meaning their capacity and bandwidth are subject to the core that owns them. Unlike this, the last-level cache is usually shared among all cores on the die. The capacity of the last-level cache is orders of magnitude larger than that of the lower-level caches. This improves efficiency for cooperative processes running on different cores, but becomes a contention point for unrelated processes.

Similarly, the bandwidth between the cores and the system's main memory is shared. Interconnects, which link sockets, chiplets, or nodes, are also shared data paths and hence can cause contention during high demand. On the lower end, hardware prefetchers are shared resources [11], in the sense that their prefetching capability can be saturated by the finite memory bus, leading to performance degradation.

Multitenancy extends the scope from a silicon level to the infrastructure level. In public clouds, a single instance of software or hardware serves multiple distinct and unrelated tenants. On the hardware side, this concerns network interface cards, storage I/O, and accelerators. These are typically connected via a PCIe bus with a finite number of lanes. In the context of container virtualization, the host kernel is shared among all guest containers. This can create

contention on kernel-internal structures [12] or entropy pools. Most other software contention points are typically specific to an application, e.g., a connection pool to a shared database.

2.2 Intel Resource Director

The Intel Resource Director Technology is a framework featured in modern Intel server hardware. It includes Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA), which allow operators to enforce partitioning of shared resources. The Intel Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM) exposed monitoring data about specific workloads to the user via the `pqos` [13] utility or directly via the `resctrl` pseudo-filesystem, which can be seen in Figure 2.1. RDT can map process IDs or cores to custom RDT Classes of Service (CLOS). The first level represents the `system/default` CLOS. The `schemata` file includes a specific mapping of resources, represented as a memory bandwidth guarantee of 10–100% and a bitmask corresponding to the cache ways available to workloads in this CLOS.

An example of such a bitmask is shown in Listing 2.1. The bitmask needs to be contiguous. For example, 110000000011_2, so the highest and lowest cache ways are not a valid bitmask on Intel hardware, but rather either 000000001111_2 or 111100000000_2. The files `cpus` and `cpu_list` can be used to select a per-socket bitmask of affected cores or a range. For example, `ffffffff` for all cores on a 32-core machine or `0-3` for the cores with ID 0 to 3. The `mode` file contains either `shareable` or `exclusive`. It determines whether the allocation group allows resource sharing. If set to exclusive, the allocation must first be relinquished, or its mode changed to allow interleaving with other bitmasks. An example is interleaving bitmasks, where a group may share a partition or parts of it, which is only possible for shared mode.

The `mon_data` and `mon_groups` directories contain files which enable access to the monitoring capabilities of RDT. They are split into domains for the available sockets and contain a file for each event. For example, `llc_occupancy` reads the bytes occupied by this CLOS. The monitoring group provides these numbers for the entire set of PIDs that are part of it. The `tasks` file includes all PIDs that are part of a given monitoring or control group. Adding a new task to a control group will remove it from its old control monitoring group. Similarly, if the tasks were previously owned by a pure monitoring group, they will be added to the new control monitoring group. If the new group is a monitoring group, it must already be moved to its parent control group when it is moved.

The `info` file contains subdirectories that hold information about the system, e.g., the number of cache ways or the maximum number of distinct CLOS groups. The custom CLOS groups are hence created under the `system/default` root level. In Figure 2.1, the CLOS `custom` was manually added. Its structure resembles the parent's structure except for the files which are only available at the root level, e.g,. the `info` directory.

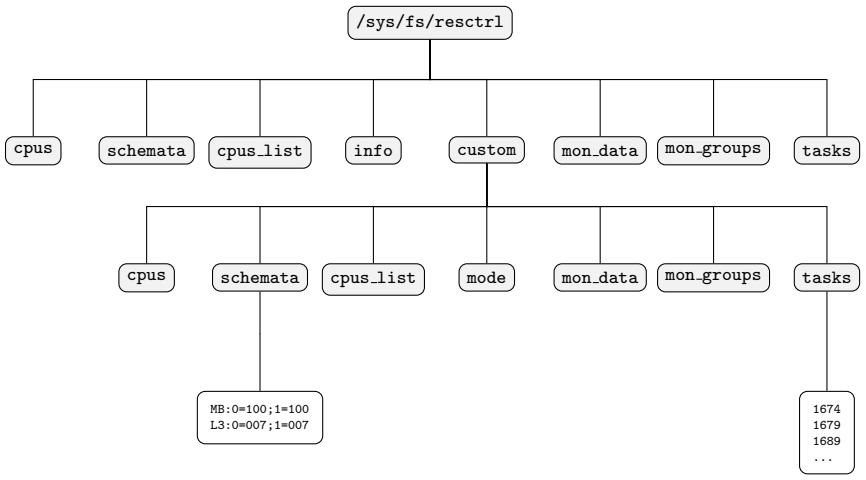


Figure 2.1: Overview of the `resctrl` filesystem structure.

```
MB:0=20;1=100
L3:0=038;1=1c0
```

Listing 2.1: An example of a `schemata` file on a CPU with a 12-way cache. The `MB` entries regulate the available bandwidth on the respective socket zero (20%) and one (100%). The two `L3` hexadecimal numbers encode bitmasks which correspond to the accessible last-level cache ways of this CLOS group. In this case, $038_{16} = 000000111000_2$ and $1c0_{16} = 000111000000_2$. This limits the CLOS to the cache ways 3–5 on socket zero and 6–8 on socket one.

Processor	**Cores**	**Last-level Cache (MB)**	**Year**
Xeon Silver 4314 [14]	16	24	Q2'21
Xeon Platinum 8360Y [15]	36	54	Q2'21
Xeon Gold 6326 [16]	16	24	Q2'21
Xeon Platinum 8480+ [17]	56	105	Q1'23
Xeon Gold 5512U [18]	28	52	Q4'23
Xeon Gold 6526Y [19]	16	37	Q4'23
Xeon Gold 6548Y+ [20]	32	60	Q4'23
Xeon Gold 6548N [21]	32	60	Q4'23
Xeon Max 9462 [22]	32	75	Q1'23

Table 2.1: Selection Intel CPUs with Resource Director Technology support, for example, the Xeon Platinum 8480+, which is used in the MareNostrum 5 [23] supercomputer.

2.3 Related Work

In this section, we introduce the current and former research on the topic. Previous related work is often built around large-scale resources such as warehouse computers or multi-site clusters. Even though we consider only smaller, homogeneous resource pools, the prior work remains relevant. It shows the previous limitations that can be overcome using the techniques introduced in this project, e.g., dynamic RDT allocation or isolated online interference probing.

2.3.1 Noisy Neighbor

The prevailing assumption typically was that the thrashing of cache lines by co-located threads was the main source of interference. Consequently, mitigation strategies focused heavily on cache partitioning and maximizing constructive cache sharing. However, Zhuravlev et al. [24] demonstrated that contention for the memory controller and memory bus often outweighed cache capacity effects. Based on this finding, the authors formalized a new "Pain" metric that quantifies the expected degradation of a co-location. The metric is derived from *sensitivity*, the likelihood that a previous cache hit will turn into a miss under contention, and *intensity*, which describes how aggressively a thread uses shared resources. To mitigate this, many contributions employ increasing synthetic contention and sensitivity profiling to categorize and co-locate defined workloads.

The Bubble-Up methodology of Mars et al. [25] addresses the over-provisioning problem in Warehouse Scale Computers (WSCs). Operators typically disallow co-locating high-priority latency-sensitive tasks with any other workload to avoid unpredictable quality of service violations. Bubble-Up proposed a mechanism to precisely predict the amount of pressure a sensitive application could tolerate. The methodology employs a tunable microbenchmark. This so-called *Bubble* generates artificial contention in the memory subsystem. By running a victim application alongside the incrementally increasing Bubble, the system generates a sensitivity curve. This curve is used to identify the exact point at which performance collapses. Conversely, the system profiles potential aggressor applications to assign them a pressure score. By mapping an aggressor's pressure score to a victim's sensitivity curve, Bubble-Up can predict interference with 1–2% accuracy.

Delimitrou et al. took a similar approach and extended it with iBench [10], which also considers shared resources like network interfaces. However, profiling every new application with tools like Bubble-Up or iBench can be prohibitively expensive due to the sheer volume of unique binaries and rapid release cycles. Delimitrou and Kozyrakis addressed this scalability challenge with Paragon [26]. In this framework, the Paragon scheduler profiles new and never-before-seen workloads for a short duration (e.g., 1 minute) against a minimal set of reference workloads. A Singular Value Decomposition (SVD) method is used to reconstruct the full interference profile of workloads by finding similarities with previously seen workloads. Besides the interference potential and its impact on a workload's performance, Paragon also considers the heterogeneity of avail-

able hosts. Hence, it is aware of where this workload would run best. While the scheduling approaches discussed above are effective at avoiding interference via placement, they lack the ability to manage interference dynamically once applications are running. Further, they cannot guarantee a certain quality of service without reserving a host for a given workload. Additionally, if a workload changes behavior, schedulers must perform expensive migrations.

2.3.2 Intel Resource Director Technology

In terms of concrete usage of the Intel Resource Director Technology, Zhu et al. introduced Perph [27]. This machine-learning-based per-node agent is designed to mitigate interference from other workloads. Specifically, they use hardware-enforced isolation via RDT Cache Allocation Technology and Memory Bandwidth Allocation to further isolate long-running, critical workloads from low-priority batch jobs.

Perph uses an Online Gradient Boost Regression Tree (OGBRT) model on a per-node basis to continuously learn the non-linear relationships between multi-dimensional resource allocation and workload performance. The model, trained offline, enables warm bootstrapping, allowing the agent to make decisions before the initial jobs start arriving on the node. Online training can mitigate model degradation when applications or configurations change. The agent was integrated into the Node Manager Apache Hadoop YARN [28] to empirically evaluate its impact on QoS and performance for data-intensive workloads such as databases or streaming.

Throughput of data-streaming applications increases by 2.0× compared to native YARN isolation and 1.82× compared to pure CGroup CPU subsystems. Similarly, TPC-C database benchmarking reveals throughput improvements of 35% and 23%, respectively, against the same baselines. This comes at the cost of delaying or potentially starving batch jobs, which are reported to have an increased makespan of 11.3%. Further, the agent does not reconsider the placement of jobs once they have been started on a node, since this is subject to the YARN Resource Manager.

Chapter 3

Methodology

In this chapter, we will introduce the theoretical framework of this project. This includes the metrics used for monitoring and evaluation. Further, we introduce a definition of the sensitivity of a given workload with regard to different shared resources. We describe the process of determining the impact of different resource allocations on a workload's performance. Lastly, we define policy algorithms to optimize the evaluation criteria.

3.1 Metric Definitions

We define and use the following metrics to quantify workload performance. The metrics can be applied to a specific workload or container or aggregated globally to quantify the performance of a scheduling policy.

3.1.1 Instructions per Cycle

Instructions per Cycle (IPC) is computed as the total number of instructions committed by the container divided by the total number of CPU cycles the container executed.

$$IPC = \frac{Instructions}{Cycles} \tag{3.1}$$

3.1.2 Stalled Cycles

Stalled cycles are those in which a container was unable to make progress. Depending on the specific workload, these stalls are typically caused by pending memory requests or control-flow dependencies. Typically, there is a distinct source of stalls that determines the overall performance of a workload (the bottleneck). However, there can be interaction and intersection between the sources, e.g., in the memory hierarchy or speculative loads.

3.1.3 Theoretical Maximum Instructions per Cycle

We define the Theoretical Maximum Instructions per Cycle (THMIC) as the number of instructions that an application could commit per cycle in a theoretically perfect system. This system would feature perfect branch prediction, instruction and data prefetching, and an infinite-capacity cache. Hence, the concrete instructions per cycle would depend solely on the container's instructions. For example, a workload that uses multi-cycle integer division will typically have a lower THMIC than one that performs bitwise operations.

$$IPC_{Theoretical} = \frac{Instructions}{Cycles - Cycles_{Stalled}} \tag{3.2}$$

3.1.4 Instructions per Cycle Efficiency

Using IPC and $IPC_{Theoretical}$, we can derive how efficiently a workload is executed on a given system, meaning how close it comes to optimal execution. We can extend this metric to consider only certain stalls, e.g., those caused by the memory subsystem. The metric can be computed using the following terms:

$$IPC_{Efficiency} = \frac{IPC}{IPC_{Theoretical}} \tag{3.3}$$

$$IPC_{Efficiency} = 1 - \frac{Cycles_{Stalled}}{Cycles} \tag{3.4}$$

3.2 Probing

Information about workloads and their relationship to shared resources is provided to the schedule via online probing. This includes information about the sensitivities to different kinds of interference and performance when allocated with different combinations of resources. The results of such probings can either be considered independently or combined to derive conclusions. In this section, we introduce the two techniques with respect to the formally described $IPC_{Efficiency}$.

3.2.1 Sensitivity

Let there be S different shared resources and A different applications. Let the average baseline Instructions per Cycle Efficiency of an application be $IPCE_i^{\text{base}}$, and the IPCE of application A_i under interference of resource S_j be $IPCE_{ij}$. The baseline values $IPCE_i^{\text{base}}$ can be measured easily. To obtain specific contention values $IPCE_{ij}$, we need to create the respective interference to the related resources S_j manually during measurement. Secondly, we need a reduction function that, for each application A_i, aggregates the vector ς_i of

length $|S|$. Each entry ς_{ij} is bounded between 0 and 1 and represents the sensitivity of A_i with respect to the inference of resource S_j. Then, the sensitivity ς_{ij} is defined as:

$$\varsigma_{ij} = \min\left(1, \max\left(0, \frac{IPCE_i^{\text{base}} - IPCE_{ij}}{IPCE_i^{\text{base}}}\right)\right) \tag{3.5}$$

3.2.2 Allocation

Depending on the host system, priority jobs may require different amounts of resources to meet their $IPCE^{target}$, which represents the quality of service guarantee it was given. Allocation probing determines this point within a set of bounds. It processes a probing queue $\mathcal{Q}_{probe}$. The probing requires the host configuration $\mathcal{C}_{host}$, which includes the hardware limits of this given host. The allocations are executed using an allocator and accounting class $\mathcal{A}$, which also stores the results. The probing is configured using bounds such as the resource bounds $(L3Ways_{\min\,/\,\max}, BW_{\min\,/\,\max})$, the granularity $(L3_{step}, BW_{step})$, the total probe time T, and the threshold $IPCE^{target}$. Algorithm 1 shows the *ascending* variant where we start with the smallest possible allocation. This allocation is increased stepwise until we either find $IPCE_{current} \geq IPCE^{target}$ or fall back to the maximum. The *descending* variant seen in Algorithm 2 starts with the maximum allocation and decreases it until $IPCE_{current}$ falls below $IPCE^{target}$ to return the previous allocation.

Algorithm 1 The ascending allocation probing method.

Require: Host Config $\mathcal{C}_{host}$, Accountant $\mathcal{A}$, Probing Queue $\mathcal{Q}_{probe}$, Steps $(L3_{step}, BW_{step})$, Bounds $(L3_{\min\,/\,\max}, BW_{\min\,/\,\max})$, Total Probe Time T, Threshold $IPCE^{target}$

$\mathcal{S}_{alloc} \leftarrow \emptyset$

▷ Generate all valid combinations

for $l \leftarrow L3_{\min}$ **to** $L3_{\max}$ **step** $L3_{step}$ **do**
  for $b \leftarrow BW_{\min}$ **to** $BW_{\max}$ **step** BW_{step} **do**
    $\mathcal{S}_{alloc} \leftarrow \mathcal{S}_{alloc} \cup \{(l, b)\}$
  end for
end for

Sort $\mathcal{S}_{alloc}$ in **Ascending** order

$t_{segment} \leftarrow T/|\mathcal{S}_{alloc}|$ ▷ Determine duration per probe

▷ Iterate upwards

for each *alloc* in $\mathcal{S}_{alloc}$ **do**
  Enforce *alloc* in $\mathcal{A}$
  Sleep for $t_{segment}$
  $IPCE_{current} \leftarrow$ MeasureIPCE()
  if $IPCE_{current} \geq IPCE^{target}$ **then**
    return *alloc* ▷ Found minimal viable allocation
  end if
end for

return $\mathcal{S}_{alloc}.last$ ▷ Default to max if target never met

Algorithm 2 The descending allocation probing method.

Require: Host Config $\mathcal{C}_{host}$, Accountant $\mathcal{A}$, Probing Queue $\mathcal{Q}_{probe}$, Steps $(L3_{step}, BW_{step})$, Bounds $(L3_{\min\,/\,\max}, BW_{\min\,/\,\max})$, Total Probe Time T, Threshold $IPCE^{target}$

$\mathcal{S}_{alloc} \leftarrow \emptyset$

▷ Generate all valid combinations

for $l \leftarrow L3_{\min}$ **to** $L3_{\max}$ **step** $L3_{step}$ **do**
  for $b \leftarrow BW_{\min}$ **to** $BW_{\max}$ **step** BW_{step} **do**
    $\mathcal{S}_{alloc} \leftarrow \mathcal{S}_{alloc} \cup \{(l, b)\}$
  end for
end for

Sort $\mathcal{S}_{alloc}$ in **Descending** order

$t_{segment} \leftarrow T / |\mathcal{S}_{alloc}|$

$alloc_{prev} \leftarrow \mathcal{S}_{alloc}.first$ ▷ Initialize with Max

▷ Iterate downwards until performance breaks

for each $alloc$ in $\mathcal{S}_{alloc}$ **do**
  Enforce $alloc$ in $\mathcal{A}$
  Sleep for $t_{segment}$
  $IPCE_{current} \leftarrow$ MeasureIPCE()
  if $IPCE_{current} < IPCE^{target}$ **then**
    return $alloc_{prev}$ ▷ Return last known good config
  end if
  $alloc_{prev} \leftarrow alloc$ ▷ Update safe fallback
end for

return $alloc_{prev}$ ▷ Min allocation is sufficient

3.3 Optimization Definition

In terms of optimization, we consider two dimensions. The first is the global efficiency of the workloads. This is mainly driven by the time during which priority workloads operate at or above their efficiency potential $IPCE^{target}$. Secondly, we consider the penalty for non-priority jobs, which is reflected in the batch job's overall IPCE. The decisions that have to be made include where to place jobs, meaning on which socket, the amount of RDT resources a job should get, and when to engage with the execution, e.g., to probe a workload.

3.4 Dynamic Scheduling Policy

The dynamic scheduling policy is shown in Algorithm 3. It incorporates probing and isolation to exploit nonlinear relationships between resource partitions of different sizes and their performance impacts. The policy processes a job queue $\mathcal{Q}_{jobs}$, which is local to a host and includes both priority and batch jobs. The priority flag is a hint to the scheduler that this workload can run significantly better with a certain amount of exclusive resources. Additionally, the priority workloads can provide a concrete $IPCE_{Efficiency}$ value, which is typically a local or global maximum for this given workload. This value is passed to the previously described allocation probing algorithms as $IPCE^{target}$ to enable early returns.

The host has a configuration, $\mathcal{C}_{host}$. It describes the availability and topology of its resources. The policy automatically adapts to the available resources to optimize the accounting $\mathcal{A}$. The accounting keeps track of how many RDT resources and cores are available and which container currently owns them. The accounting implements optimizations to efficiently relinquish and reuse resources. Priority jobs need probing to determine exactly how many RDT resources they require on this specific host, as detailed in Section 3.2.2. To this end, the priority jobs are added to a probing queue $\mathcal{Q}_{probe}$. Whenever a new probing result is available, the scheduler will likewise enqueue it in the allocation queue $\mathcal{Q}_{alloc}$ or enforce it immediately. Since probing itself requires allocations, it also adds entries to the allocation queue. To enable a consecutive, uninterrupted sweep for the required allocation probing, it will buffer a minimum amount of required allocations. The scheduler processes the $\mathcal{Q}_{alloc}$ until it encounters an infeasible request, e.g., because insufficient applicable resources are available.

For RDT cache partitions, the bitmasks used to allocate them must be contiguous. Hence, the cache ways are subject to fragmentation, which can be mitigated by reallocating them. This is implemented in the accounting consolidation function. Once the $\mathcal{Q}_{alloc}$ queue is either empty or blocked, the scheduler processes the $\mathcal{Q}_{jobs}$ queue. If sufficient cores are available at the head of this queue, it will be admitted. If this job is a priority job, it will be placed on the socket with fewer committed resources. Until the first probing of this job is available, the scheduler will assume that it needs all resources. The sched-

uler will enqueue the job for probing with a warm-up time of at least $\mathcal{T}_{warmup}$ seconds. For batch jobs, the scheduler selects the socket with the least load.

The load is estimated using a heuristic that sums the stalled cycles caused by the memory subsystem. If rebalancing is enabled, the scheduler will continuously rebalance the load. The scheduler is invoked every $\mathcal{T}_{tick}$ seconds.

Algorithm 3 The dynamic priority scheduling policy.

Require: Job Queue $\mathcal{Q}_{jobs}$, Host Configuration $\mathcal{C}_{host}$, Accountant $\mathcal{A}$, Probing Queue $\mathcal{Q}_{probe} \leftarrow \emptyset$, Allocation Queue $\mathcal{Q}_{alloc} \leftarrow \emptyset$, Interval $\mathcal{T}_{tick}$, Warmup time $\mathcal{T}_{warmup}$

while True **do**
  while there are finished jobs **do**
    Relinquish resources in $\mathcal{A}$
  end while
  while new allocation probe results exist **do**
    Enqueue in order into $\mathcal{Q}_{alloc}$
  end while
  while $\mathcal{Q}_{alloc}$ is not empty **do**
    if Infeasible($\mathcal{Q}_{alloc}, \mathcal{A}$) **then**
      Consolidate($\mathcal{Q}_{alloc}, \mathcal{A}$) ▷ Defragment
    end if
    if Infeasible($\mathcal{Q}_{alloc}, \mathcal{A}$) **then**
      break
    end if
    $req \leftarrow \mathcal{Q}_{alloc}$
    if req is Probe Allocation **then**
      Buffer allocation
      if Buffer is sufficient for Probe($\mathcal{Q}_{probe}$) **then**
        Probe($\mathcal{Q}_{probe}$) and Dequeue($\mathcal{Q}_{probe}$)
      end if
    else
      Allocate($\mathcal{Q}_{alloc}, \mathcal{A}$) ▷ Next Priority Job
    end if
  end while
  if AdmissionPossible($\mathcal{Q}_{jobs}.head$) **then**
    $J \leftarrow$ Admit($\mathcal{Q}_{jobs}.head$) ▷ No backfilling or bypassing
    if J is Priority **then**
      Place J on socket with more uncommitted resources
      Enqueue allocation for Probe($\mathcal{Q}_{alloc}$)
    else ▷ Non-Priority
      Place J on socket with least load
      if Rebalancing non-priority jobs is enabled **then**
        Start timer $\mathcal{T}_{warmup}$ for rebalance after warmup
      end if
    end if
  end if
  if Rebalance is due **then**
    Rebalance
  end if
  Sleep for $\mathcal{T}_{tick}$ seconds
end while

3.5 Least Loaded Policy

Least Loaded is a common heuristic for scheduling jobs. In the context of a dual-socket node, the least loaded policy, as seen in Algorithm 4, assigns arriving jobs to the socket with the currently least committed compute cores. The job queue $\mathcal{Q}_{jobs}$ is processed strictly sequentially, meaning there are no attempts to backfill any job. Further, the policy does not distinguish between critical priority jobs and batch jobs. Depending on the order of arriving jobs, however, the policy can decrease interference by potentially balancing the load across the sockets.

Algorithm 4 The least-loaded scheduling policy.

Require: Job Queue $\mathcal{Q}_{jobs}$, Host Configuration $\mathcal{C}_{host}$ (containing sockets $\mathcal{S}$), Accountant $\mathcal{A}$, Interval $\mathcal{T}_{tick}$

while True **do**
  while there are finished jobs **do**
    Relinquish resources in $\mathcal{A}$
  end while
  while $\mathcal{Q}_{jobs}$ is not empty **do**
    $J \leftarrow \mathcal{Q}_{jobs}.head$
    $s_{best} \leftarrow \emptyset$
    $L_{\min} \leftarrow \infty$
    for each socket $s \in \mathcal{S}$ **do**
      $L_{current} \leftarrow$ GetCommittedCores$(s, \mathcal{A})$
      if $L_{current} < L_{\min}$ **and** Fits(J, s) **then**
        $L_{\min} \leftarrow L_{current}$
        $s_{best} \leftarrow s$
      end if
    end for
    if $s_{best} \neq \emptyset$ **then**
      Allocate$(J, s_{best}, \mathcal{A})$
      Dequeue$(\mathcal{Q}_{jobs})$
    else
      break ▷ Insufficient resources, wait for next tick
    end if
  end while
  Sleep for $\mathcal{T}_{tick}$ seconds
end while

Chapter 4

Implementation

In this work, we aim to monitor, study, and mitigate the noisy neighbor effect at the container level. To test theoretical models and decision-making policies, we implemented a comprehensive experimental setup. This setup covers all aspects and metrics relevant to our benchmarks and beyond. It enables complex experimental scenarios and, most importantly, integrates the container runtime and Intel Resource Director into a robust and dynamic tool chain. On a higher level, it bundles the following key aspects:

- Benchmark Orchestration
- Automatic Collection for Analysis and Live Decision Making
- Decision Making Interface
- Robust and reliable RDT Interface
- Visualized Analysis Framework

4.1 Requirements

In the following sections, we introduce the requirements that we identified for such a system. Further, we present the technical implementation and provide detailed descriptions of its components and optimizations.

4.1.1 Orchestration

Coordinating benchmarks in which we want multiple independent container jobs to arrive and run according to a set of parameters requires a central, flexible orchestrator. We want to support a variety of benchmarking applications. Hence, we need a unified way of integrating these into our benchmark pool. From this, we either want to manually build traces to test concrete noisy-neighbor scenarios and observe how our decision-making behaves, or generate a trace on demand for long-running experiments configured at a higher level.

4.1.2 Instrumentation

We need to collect all relevant container metrics to gain insight into the ongoing experiment and to inform optimization decisions. This includes metrics from the applications running inside the container itself, e.g., hardware counters, but we also need to track our own decisions, e.g., where a container job was placed or how many L3 cache ways were allocated during that period. Since we want to enable decision-making that operates in near-real time, we need to support higher-than-usual sampling rates.

4.1.3 Decision Making

A central scheduler needs to make decisions to optimize one or more metrics, e.g., the number of cache misses. To this end, the scheduler needs the relevant information for its decision-making. Likewise, the scheduler needs interfaces to realize these decisions, e.g., to allocate exclusive L3 cache ways using Intel RDT.

4.1.4 Instrumentation Interference

The experimental setup must support a maximum-load scenario, e.g., where each core runs a highly data-intensive container application. However, the memory footprint and, hence, the interference from the experimental setup itself need to be low to avoid disturbing the benchmark results.

4.2 Architecture

The higher-level architecture meets the defined requirements and also enables future multi-node support. This is achieved by having multiple completely independent running modules. This includes the orchestrator autonomously configuring and starting containers and enqueuing them for admission. The orchestrator also launches a configurable data collector per container, which will run in a separate process. The data collector uses various interfaces, such as runtime sockets or hardware counters, to periodically sample data on the container's workload and current configuration.

The data is made available to a scheduler, which can analyze it to make decisions. The scheduler has access to an allocator which bundles resource-related actions, e.g on which core a container should be allocated or to which RDT class it should belong. The allocator automatically keeps track of currently allocated resources. The scheduler also has access to a prober. This probe bundles a mechanism to asynchronously test a container regarding its interference sensitivity to shared resources. Internally, it can also use the allocator, e.g., to find an acceptable allocation for a given container and return it to the scheduler.

4.3 Orchestrator

The orchestrator integrates and configures all other components. It consists of a front end that parses all possible configurations for container arrival distributions, image sources, the database connection, the collectors, the scheduler initialization, and job feeding during arrivals, and performs teardown at the end of a benchmark. All of these aspects can be defined in a configuration file.

Listing 4.1 shows the upper part of such a file, which concerns the metadata such as the name and description of the benchmark, as well as higher-level parameters. The `max_t` field tells the orchestrator to stop the experiment after thirty seconds, regardless of the current job queue. In the `scheduler` section, we can select a particular scheduler implementation and specify secondary parameters, such as whether the scheduler should be initialized with an RDT allocation interface. The `prober` subsection allows the user to select a probing implementation and a default configuration to use when the scheduler issues a container probe. The `data` section configures the database client used to write results to a persistent database after the benchmark finishes. In the `docker` section, we can log in to a private container registry so the orchestrator can pull the container images we prepared for the benchmark.

```yaml
benchmark:
  name: "Example Benchmark"
  description: "Description of the benchmark"
  max_t: 30
  log_level: info
  scheduler:
    implementation: default
    rdt: false
    log_level: warn
    prober:
      implementation: default
      isolated: true
      default_t: 15
      warmup_t: 10
  data:
    db:
      host: ${INFLUXDB_HOST}
      name: ${INFLUXDB_BUCKET}
      user: ${INFLUXDB_USER}
      password: ${INFLUXDB_TOKEN}
  docker:
    registry:
      host: ${DOCKER_REGISTRY_HOST}
      username: ${DOCKER_REGISTRY_USERNAME}
      password: ${DOCKER_REGISTRY_PASSWORD}
```

Listing 4.1: An example of a simple benchmark configuration file.

Listing 4.2 includes an example of a static trace definition, meaning the

user decides what container is started when and how long it is running. The examples showcase all of the available configuration options. However, most options default to the same value, so the actual file can be more compact. Each container can have specific configurations. For example, `example-container` can map a port to the host in case we want to externally reach it for load generation. In the `command` section, we can define a shell command that the orchestrator injects into the container at startup. Similarly, the orchestrator can mount directories, set environment variables, or elevate the privileges of a given container once it is started after `start_t` seconds or by default upon the start of the benchmark.

In the `data` section, we can specify which data should be collected for each container during the benchmark and at what frequency. For the `example-container`, we will collect all the available data every one hundred milliseconds, while the `matrix-3d` container will only be sampled with a selection of the available `perf` counter at a much lower frequency.

```yaml
example-container:
  port: 8080:8080
  start_t: 20
  image: nginx:alpine
  command: "apk add stress-ng && stress-ng --cpu 1 --timeout 10s"
  environment:
    FOO: hello
    BAR: world
  privileged: false
  volumes:
    - ./data:/data
  data:
    frequency: 100
    perf: true
    docker: true
    rdt: false

matrix-3d:
  image: registry.jakob-eberhardt.de/rdt4nn/stress-ng
  command: "stress-ng --matrix-3d 1 --timeout 0 "
  data:
    frequency: 2000
    perf:
      instructions: true
      cycles: true
      stalls_total: true
      stalls_l3_miss: true
    docker: false
    rdt: false
```

Listing 4.2: An example of a static trace definition.

The second option to define and run an experiment is by configuring a statistical distribution shown in Listing 4.3, which generates a reproducible trace

of arriving jobs based on a seed from a pool of workloads, which, for example, can be seen in

Listing 4.4. The configuration covers all common aspects of the related research, including the mean inter-arrival time of jobs and their respective standard deviations, the distribution of workload duration, and a split by job type. In our setup, we distinguish between single- and multi-threaded workloads, multi-programming, and IO-bound workloads. Based on these parameters, the orchestrator will generate a deterministic trace that can be rerun with, for example, different scheduler implementations. Since we know which images and containers will be required, we can prefetch the images and create the containers in advance. This reduces orchestration overhead during execution, since starting a pre-created container is lightweight. The orchestrator also sets up a network so that containers can reach each other via their container names, e.g., when multi-programmed jobs need to interact. The expected interference can be tuned via the `sensitivities` split of the jobs.

```yaml
arrival:
  seed: 123
  mean: 10
  sigma: 2
  length:
    mean: 60
    sigma: 20
    min: 10
  split:
    single-thread: 50
    multi-programmed: 10
    multi-thread: 30
    iobound: 10
  sensitivities:
    low: 60
    medium: 0
    high: 40

data:
  frequency: 10000
  perf: true
  docker: true
  rdt: true

workloads:
  input:
    - ./jobs/stress-ng
    - ./jobs/compression
    - ./jobs/ffmpeg
```

Listing 4.3: An example of an automatic trace generation configuration file.

The jobs can be organized in a hierarchy, e.g, by application and sub-configurations. The workload definitions include the required `image` and an

optional `command` which will be executed. Jobs can request a fixed amount of cores, a range, or a set of core counts. Valid `num_cores` include for example `2`, `2-4,6`, or `1,2,3,12`. Each job is labeled with a `kind` and `sensitivity`. Note that the scheduler is not provided with this information.

```yaml
workloads:
  matrix-3d-default:
    image: registry.jakob-eberhardt.de/rdt4nn/stress-ng
    command: "stress-ng --matrix-3d 1 --timeout 0"
    num_cores: 1
    kind: single-thread
    sensitivity: high

  matrix-3d-small:
    image: registry.jakob-eberhardt.de/rdt4nn/stress-ng
    command: "stress-ng --matrix-3d 1 --matrix-3d-size 8 --timeout 0"
    num_cores: 1
    kind: single-thread
    sensitivity: low
```

Listing 4.4: An example pool of jobs to select from. The same application can show different sensitivities depending on its configuration, e.g., here the size of the 3D matrix.

4.3.1 Container Runtime Socket

The orchestrator integrates the container runtime via the Docker Engine API [29, 30]. This includes creating the container according to the configuration seen in Listing 4.5. In addition to orchestrating container creation, start, and stop, the orchestrator continuously monitors the benchmark and detects unexpected events, such as unplanned container exits or error codes. It also validates the execution plan and logs the relevant information, for example, if an explicit core affinity mapping is infeasible on the given host.

At container start, the orchestrator uses the Docker API to bootstrap other components, for example, to obtain the root PID of a new container, which must be passed to the RDT manager. Similarly, we can obtain the container's CGroup, which is needed to initialize the container's hardware counter collector.

```
type ContainerConfig struct {
    Index       int              `yaml:"index"`
    Name        string           `yaml:"name,omitempty"`
    Image       string           `yaml:"image"`
    Port        string           `yaml:"port,omitempty"`
    Core        string           `yaml:"core,omitempty"`
    NumCores    int              `yaml:"num_cores,omitempty"`
    CPUCores    []int            `yaml:"-"`
    StartT      *int             `yaml:"start_t,omitempty"`
    StopT       *int             `yaml:"stop_t,omitempty"`
    ExpectedT   *int             `yaml:"expected_t,omitempty"`
    Command     string           `yaml:"command,omitempty"`
    Privileged  bool             `yaml:"privileged,omitempty"`
    Environment map[string]string `yaml:"environment,omitempty"`
    Volumes     []string         `yaml:"volumes,omitempty"`
    Data        CollectorConfig  `yaml:"data"`
}
```

Listing 4.5: The internal container configuration structure.

4.4 Data Collectors

The experimental setup features a variety of metrics that can be collected during a benchmark. This includes data on the container's execution state and the host. In this section, we will introduce the technical solutions we developed to enable robust, scalable, high-frequency sampling across many concurrently running containers. Even though most data sampling will be conducted at low frequency, we want to adapt it dynamically, e.g., to detect micro patterns. To this end, we have to integrate different sources that can work internally in drastically different ways. These include the Docker socket for higher-level data, such as the CPU usage or the used memory, perf hardware counters, e.g., to compute the instructions a container is processing per cycle, and metrics that are exclusive to the RDT framework, such as the last-level cache occupancy or the used memory bandwidth.

As mentioned, each tool is exposed to the programmer in a different way. Integrating these different sources into a robust application and scaling it to many simultaneously running containers requires concurrent programming. Some data endpoints operate asynchronously, such as the Docker socket, while others require synchronous actions, for example, opening and reading a perf event counter.

The resulting thread-safe data structure is introduced in Subsection 4.4.1. The integration and capabilities of the RDT monitoring features are described in Section 4.4.2. The technical measures for monitoring the hardware counters of a container and, hence, its CGroup are explained in Subsection 4.4.3. The Docker metrics integration is introduced in Section 4.4.4. The metadata used for experimental analysis is covered in Subsection 4.4.6.

4.4.1 A Thread-Safe Datastructure

For each container, we organize the data collected during the benchmark execution in a `DataFrame`, as shown in Listings 4.6 and 4.7. This structure includes all information that might be relevant for a scheduler implementation or a manual analysis of a benchmark. During a benchmark, multiple independent processes, such as collectors or schedulers, will access the data. To avoid race conditions and runtime panics, the data structure provides synchronized functions for safe access. To this end, we employ `RWMutex` locks, which allow readers to proceed in parallel while writers can obtain exclusive access to the `container` that exposes the currently running containers.

```
type DataFrames struct {
    containers map[int]*ContainerDataFrame
    mutex      sync.RWMutex
}
```

Listing 4.6: The `Dataframes` structure, which is periodically populated by the collectors and passed to the scheduler.

Since each container will have collectors running different configurations and sampling frequencies in separate processes, the `ContainerDataFrame` structures have individual locks that synchronize access to their respective `steps` maps at a finer granularity. As shown in Listing 4.10, the `ContainerDataFrame` itself aggregates the three data substructures `Perf`, `Docker`, and `RDT`. Although synchronizing the sampling on this level would likely further increase the maximum sampling rate at the cost of more locks, the gained frequencies have no practical relevance in the scope of this project, as it is already more than enough, as mentioned in Section 4.4.5. However, relying solely on global locking, i.e., only on the `containers` map, is infeasible and can lead to excessive contention for the lock, especially for writers such as collectors that want to populate the dataframes of different containers.

An example of a synchronized read access is shown in Listing 4.8, where we use a `RLock` read-only lock in the `GetLatestStep` function to obtain the latest

available sampling step for a given container `cdf`. This function is typically used by scheduler implementations to obtain information about a container without having to retrieve all of the previous steps, which would pollute the cache. The frame is populated by the synchronized `AddStep` function seen in Listing 4.9.

```
type ContainerDataFrame struct {
    steps map[int]*SamplingStep
    mutex sync.RWMutex
}
```

Listing 4.7: A particular `DataFrame` which consists of the currently available sampling steps and a lock needed to access it.

```
func (cdf *ContainerDataFrame) GetLatestStep() *SamplingStep {
    cdf.mutex.RLock()
    defer cdf.mutex.RUnlock()

    maxStep := -1
    var latest *SamplingStep
    for step, data := range cdf.steps {
        if step > maxStep {
            maxStep = step
            latest = data
        }
    }
    return latest
}
```

Listing 4.8: The `GetLatestStep` function, implemented by the `ContainerDataFrame` structure, returns the latest sampling for a specific container. To this end, it requests a read-lock to safely handle the request. When iterating over the `steps` map, we only need to pull the step integer and the `SamplingStep` pointer into cache. The `defer` keyword releases the read lock upon return or failure.

```
func (cdf *ContainerDataFrame) AddStep(stepNumber int, step
    *SamplingStep) {
    cdf.mutex.Lock()
    defer cdf.mutex.Unlock()
    cdf.steps[stepNumber] = step
}
```

Listing 4.9: The `AddStep` function is part of the `ContainerDataFrame` structure and is used to add new data to a container after it has finished scraping. To do this safely, it uses a write lock.

The collectors for each data source can be configured individually, allowing us to select fields and sampling frequencies for each source. This further reduces the memory footprint of the experimental setup for certain metrics. For example,

CPU usage from the Docker socket can be used to monitor a container's activity. Typically, it can be queried at a lower frequency than, for example, the last-level cache occupancy metric of RDT. The full structures, including all available data fields, are shown in the appendix Listings A.1, A.2, and A.3. In the following, we present a subset of each counter.

```
type SamplingStep struct {
    Timestamp time.Time
    Perf      *PerfMetrics
    Docker    *DockerMetrics
    RDT       *RDTMetrics
}
```

Listing 4.10: The `SamplingStep` structure.

```
type RDTMetrics struct {
    RDTClassName *string
    MonGroupName *string
    MBAThrottle *uint64

    // Per-socket monitoring metrics
    L3OccupancyPerSocket       map[int]uint64
    MemoryBandwidthLocalPerSocket map[int]uint64
    L3UtilizationPctPerSocket  map[int]float64
    MemBandwidthMbpsPerSocket  map[int]float64

    // Per-socket allocation details
    L3BitmaskPerSocket     map[int]string
    L3WaysPerSocket        map[int]uint64
    MBAPercentPerSocket    map[int]uint64
}
```

Listing 4.11: The `RDTMetrics` structure. The `RDTClassName` represents the CLOS group the container was a member of at the time of the sampling. Similarly, `MonGroupName` tracks the RDT monitoring group. Certain metrics, such as the last-level cache occupancy, are provided per socket; hence, we map them to the respective CPU. The collector tracks allocation settings independently of the scheduler to detect temporal deviations, e.g., when an allocation fails.

```
type PerfMetrics struct {
    Instructions     *uint64
    Cycles           *uint64
    CacheMisses      *uint64
    CacheReferences  *uint64
    BranchInstructions *uint64
    BranchMisses     *uint64

    StallsTotal            *uint64
    StallsL3Miss           *uint64
    StallsL2Miss           *uint64
    StallsL1dMiss          *uint64
    StallsMemAny           *uint64
    ResourceStallsSB       *uint64
    ResourceStallsScoreboard *uint64

    // Derived metrics
    CacheMissRate      *float64
    InstructionsPerCycle *float64
    StalledCyclesPercent *float64
    StallsL3MissPercent *float64
    TheoreticalIPC     *float64
    IPCEfficiency      *float64
}
```

Listing 4.12: The `PerfMetrics` structure holds data that is sourced from the hardware counter. Most importantly, these include the `Cycles` an given container was using the CPU and the `Instructions` it executed within the sampling interval. Since we are particularly interested in shared resources, we can track the cycles in which the execution was stalled by a shared resource, such as the last-level cache (`StallsL3Miss`) or the percentage of the total stalls that were caused by pending memory requests due to LLC misses (`StallsL3MissPercent`).

```
type DockerMetrics struct {
    CPUUsagePercent  *float64
    CPUThrottling    *uint64
    MemoryUsage      *uint64
    NetworkRxBytes   *uint64
    NetworkTxBytes   *uint64
    DiskReadBytes    *uint64
    DiskWriteBytes   *uint64
    AssignedCores  map[int]bool
}
```

Listing 4.13: The `DockerMetrics` structure which reflect for example the output of the `docker stats` command.

During benchmarking, potential noise from other system processes or the experimental setup can be isolated using RDT. To this end, we alter the `system/default` group and move every container into an isolated benchmark group upon its arrival. We can estimate the memory footprint of each sampling step in the worst case, where all collectors and their fields are enabled on a 64-bit system. Note that this is an intentionally pessimistic upper bound: in typical benchmark configurations, only a subset of collectors is active, and even when active, many optional fields remain unset (`nil`) and therefore do not allocate additional objects.

At the data structure level, a sampling step consists of a small header (a timestamp plus three pointers to metric structs) followed by the metric payloads for `Perf`, `Docker`, and `RDT`. While the metric values themselves are mostly scalar counters (64-bit integers or floats), the dominant cost in Go is often not the scalar payload but the number of heap objects and the associated bookkeeping:

1. Many fields are stored as pointers (e.g., `*uint64`, `*float64`), which implies separate heap allocations when populated
2. Per-socket maps (primarily in `RDT`) allocate buckets
3. String fields (e.g., allocation schemata, bitmasks, CPU-set) contribute variable-sized byte arrays

We therefore treat the fixed-size struct headers as a constant term and focus the estimate on map and string-heavy components. We assume a constant ≈ 0.5 KB for the base struct headers (`Timestamp`, `Perf`, `Docker`, `RDT`) and then add an estimate for the fully-populated metric data. As shown in Table 4.1, the raw dataframe contains 59 fields in total: 11 derived (e.g., cache miss rate, IPC efficiency) and 48 direct readings.

The `RDT` struct is the main contributor in the worst case because it contains multiple per-socket maps and allocation strings scraped from the resctrl configuration. With all collectors enabled and assuming a dual-socket host, our estimate is $\approx$ 3–10 KB per container and sampling step. Hence, we can estimate the memory complexity with the following formula:

$$extMemory \approx (\#\text{containers}) \times (\#\text{steps per container}) \times (3\text{–}10\,\text{KB}). \quad (4.1)$$

Struct	Field Count	Derived	Raw
Perf	28	6	22
Docker	15	2	13
RDT	14	3	11
Total	59	11	48

Table 4.1: An estimation of the worst-case memory footprint of the data collection.

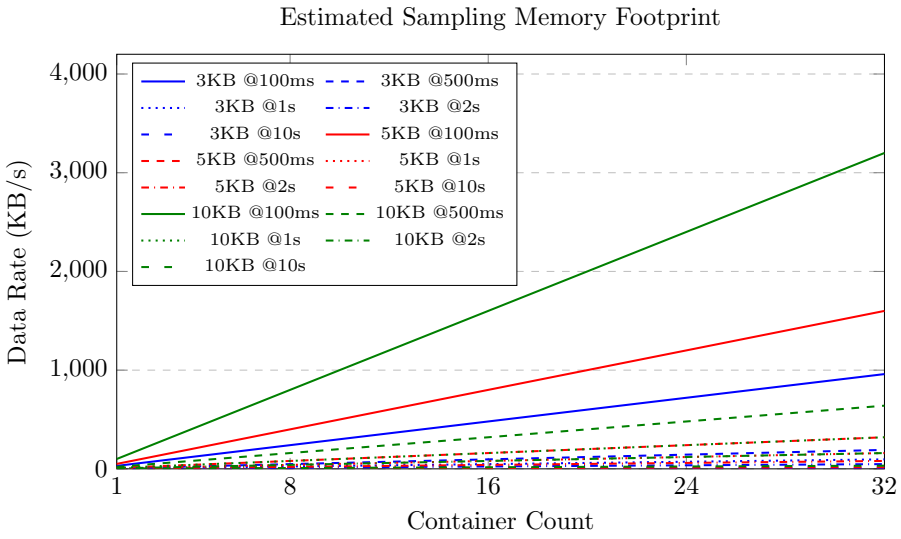


Figure 4.1: The data used per second under different sampling rates and container counts for collecting all fields. In practice, fewer counters are enabled than the available 59. The noise can also be eliminated using RDT.

In practice, we typically use a sampling interval of two to ten seconds and temporarily increase it for a given container, e.g., for detailed analysis. In any case, the memory footprint is acceptable for two reasons. First, it is a worst-case scenario: most runs either disable some collectors entirely or record only a subset of fields, leaving many pointers unset and reducing map and string allocations.

Second, the hot-path access patterns are typically dominated by recent data. Schedulers and probes usually use only the most recent step (or a short window) for decision-making. Iterating over steps (e.g., to find the latest step by step number) touches the map keys and pointers, but does not necessarily dereference and pull the full metric payload into cache. The larger metric structs and their

maps only become cache-relevant when their values are actually read. The full history is primarily needed for post-processing and database export after the benchmark, where the setup can no longer interfere with the experiment.

4.4.2 RDT Allocation & Monitoring

To integrate RDT's monitoring capabilities, the experimental setup must, by default, create an explicit monitoring group (`mon_group`) for each container, which is synchronized with the tasks running inside the respective container. This enables filtering for events belonging to a single container rather than, for example, the `system/default` group. Likewise, if the scheduler added a container to CLOS including the default group, the collector needs to safely create a new control monitoring group (`ctrl_mon`) under this control group using the library function `CreateMonGroup` [31] and migrate the container's PIDs to it.

Since each RDT collector runs in a separate process, access to the RDT interface must be synchronized, as shown in Listing 4.14. Containers may spawn new processes that require periodic synchronization with the PIDs sharing the container's root process's CGroup. Note that the RDT integration with the underlying low-level container runtime, runc [32], is under active development. Hence, the runtime itself could handle this task in the future. One design principle of the experimental setup is the separation between the different modules: the scheduler is unaware of the monitoring group changes the collector performs, and likewise, the collector discovers CLOS migrations once they are executed by the scheduler's RDT interface. This way, we can evaluate and develop the scheduler independently of its internal accounting, which can be affected by failed allocations or jitter. We see exactly what happened during the benchmark with respect to the resource-control filesystem. The structure of the monitoring part of the `resctrl` filesystem can be seen in Figure 4.2. Its root `container_test_mon` refers to the name of the container it was created for.

As already mentioned in Section 2.2, the files `cpus` and `cpu_lists` contain bitmasks and ranges for CPU cores that should be used when filtering RDT monitoring events. Likewise, the `tasks` file includes the container's root PID and its child processes, which are determined by its CGroup. The `mon_data` directory contains subdirectories for the two sockets of the system. The files contain the filtered events, such as the `llc-occupancy` in bytes. The memory bandwidth metrics are separated in `mbm_local_bytes`, which measures only the data transferred between the cores and the local memory controller, and `mbm_total_bytes`, which includes data regardless of where the memory is located.

Intel published an errata [33] for certain Skylake-generation hardware, which causes the MBM counters to report incorrect data. If a task, or, in this case, a container, is moved to a new group, its monitoring counter will start from zero, but the lines in cache from the old group remain valid and continue to contribute to its occupancy.

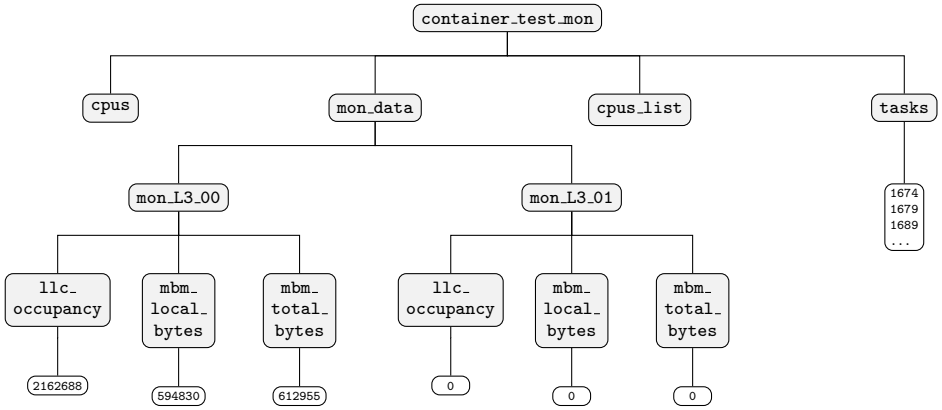


Figure 4.2: The filesystem structure of a monitoring group in `resctrl`.

```
rdtguard.Lock()
monGroup, err := ctrlGroup.CreateMonGroup(
    monGroupName, map[string]string{
       "container-bench": "true",
       "pid":             pidStr,
})

rdtguard.Unlock()
if err != nil {
    return nil, fmt.Errorf("failed to create RDT monitoring group: %v",
        err)
}
```

Listing 4.14: Creating a monitoring group for a benchmark container. In combination with PID syncing, this enables container-level RDT monitoring.

4.4.3 Hardware Counters

In this project, we use perf [34, 35] to access low-level hardware counters to map them to containers running in a benchmark. In particular, we are interested in counters related to shared resources, such as the number of cache misses or the total execution stalls caused by last-level cache misses. However, the setup also integrates other metrics, e.g., branch misprediction. Many Intel processors feature the Top-down analysis with perf [36], a framework that aggregates metrics to identify bottlenecks.

Unfortunately, the tool does not support CGroup filtering, which is essential for collecting data on entire containers rather than a single running binary.

Further, we want to periodically sample the container and provide the scheduler with new data. To achieve this reliably at high sampling rates and with many concurrently running containers, we need to efficiently manage access to the hardware counters, which are available via a finite number of Performance Monitoring Units (PMUs). While opening perf events directly enables CGroup filtering, many counters support only CPU-core-specific filtering. Hence, we need to cross-check which cores the container is currently assigned to, then open and read the events of its respective CGroup on each of those cores. This can increase pressure on the PMUs, especially when sampling at high frequencies and when a large number of cores are assigned to containers. If the PMUs are oversubscribed, the kernel will start multiplexing them to serve incoming requests. Effectively, the units are time-shared by reducing the time per event.

However, the kernel also reports how long the event was enabled and running. Hence, we can adjust the total sum when multiplexing is used, as shown in Listing 4.15. During perf collector initialization, all configured events are opened. The loop processes the events and reads their counts using the `ReadCount` library function. The `count` struct contains the `Value` field, which is the raw cumulative count of events. The `Enabled` is the total time the event was enabled on the PMU, and `Running` is the time it actually ran on a hardware counter. By calculating the deltas with respect to the last step, we can detect if the event was multiplexed and, if so, apply a scaling factor and correct the `deltaValue` with the `scaleFactor` such that we can compute:

$$\text{scaleFactor} = \frac{\Delta\text{Enabled}}{\Delta\text{Running}} \tag{4.2}$$

$$\text{scaledDelta} = \Delta\text{Value} \times \text{scaleFactor} \tag{4.3}$$

```
for i, event := range pc.events {
        count, err := event.ReadCount()
        if err != nil {
            continue
        }

        // Get the current cumulative values
        currentValue := uint64(count.Value)
        currentEnabled := count.Enabled
        currentRunning := count.Running

        // Calculate deltas from the last sample
        if lastState, exists := pc.lastState[i]; exists {
            deltaValue := currentValue - lastState.value
            deltaEnabled := currentEnabled - lastState.enabled
            deltaRunning := currentRunning - lastState.running

            // Apply multiplexing correction
            scaledDelta := deltaValue
            if deltaRunning > 0 && deltaEnabled > 0 && deltaRunning !=
                deltaEnabled {
                // Scale the delta by ratio
                scaleFactor := float64(deltaEnabled) /
                    float64(deltaRunning)
                scaledDelta = uint64(float64(deltaValue) * scaleFactor)
            }

            counterSums[count.Label] += scaledDelta
        }

        pc.lastState[i] = &eventState{
            value:  currentValue,
            enabled: currentEnabled,
            running: currentRunning,
        }
}
```

Listing 4.15: Handling the multiplexing of PMUs by applying a scaling factor if needed.

The values of each event are aggregated across the cores where the container is currently running. Lastly, the results are added to the `PerfMetric` of the dataframe struct. The perf collector implementation works independently of a specific target host, meaning it can be used on any Intel architecture that supports the configured hardware counter, including consumer hardware.

4.4.4 Docker Metrics

The Docker collector uses a dedicated client to access the native runtime metrics of the containers. Unlike the standard `ContainerStats`, the `OneShot` endpoint for metrics will not wait to prime container metrics but rather returns immediately when the stats are available. This is better suited to our use case, as it allows us to maintain our native sampling frequency and aggregate the data ourselves.

4.4.5 Sampling Rate

All available counters are integrated into the setup, supporting an adaptive sampling rate of up to 10 Hz (every 100 milliseconds) for fully fledged data-taking with more than 30 concurrently running containers, and up to 50 Hz (every 20 milliseconds) for up to 6 concurrently running containers. Note that the amount of scraped data can be adjusted to reduce the setup's memory footprint or enable higher sampling frequencies for more containers.

4.4.6 Metadata

The experiment driver keeps track of the relevant metadata during a benchmark run. This includes the host configuration with details of the microarchitecture (number of cores, topology, cache sizes, and ways), but, more importantly, it also independently tracks the scheduler's decisions. This enables transparency, e.g., a track record of which container ran on which socket or core at a given time. From this, we can derive how well a scheduler implementation balances the pressure. Additionally, the application uploads the benchmark configuration and the used trace to the database, enabling easy reproduction or reconfiguration.

4.5 Probing

During the execution, the scheduler needs to know which container is sensitive to which kind of interference. Similarly, the scheduler needs a consistent, easy-to-use way to determine how many RDT resources to add to a container so it can run according to its priority. To this end, we encapsulated these processes in `Probes`, which can be configured and launched by the scheduler.

4.5.1 Allocation Probing

The allocation probing aims to find the first valid allocation that meets the container's priority guarantee. Similar to the sensitivities, `AllocationProbe` bundles multiple steps into a well-defined process that returns the required `AllocationProbeResult` object, which resembles a set of RDT resources that are autonomously and asynchronously managed. The allocation probing process is configured with different values that can be tuned for quick convergence

and penalty reduction for unrelated containers. The configuration is shown in Listing 4.16.

In general, the prober will start with the minimum allocation `min_l3` last-level cache ways and a memory bandwidth of `min_mem`. Within each way allocation, we increase the memory allocation by `mem_step` until the maximum, then reset it to the minimum and proceed to the next greater cache way allocation. We test each allocation for its fraction of the `probing_t` until we either finish all combinations or encounter a `break_condition` which defines the quality of service threshold, e.g., 90% of the best possible IPC for this container. If `greedy_allocation` is enabled, we will test all combinations and choose the one which fits better into the current allocation, e.g. if most of the memory bandwidth is already allocated in the former allocation (the one we temporarily loosen for the probing), we keep trying to find an allocation where we can compensate for this with more last-level cache ways. The limiting factors of RDT, detailed in Section 6.1, significantly increase the required allocation for most workloads.

```yaml
prober:
    min_l3: 1
    max_l3: 1
    step_l3: 1
    min_mem: 10
    max_mem: 10
    mem_step: 10
    probing_t: 10
    cooldown_t: 5
    break_condition: 0.9
    warmup_t: 20
    reverse_probe_order: false
    greedy_allocation: false
    probing_frequency: 50
    allocate: false
```

Listing 4.16: Configuration parameters of a `AllocationProbe`.

4.5.2 Sensitivity Probing

The goal of a sensitivity probe is to improve efficiency by studying the impact of nuanced interference on a specific workload. This is especially useful in multi-node setups, where one has degrees of freedom in container placement. However, achieving accurate probing was usually limited to offline probing, meaning on a separate node, where, for example, the job is started alone and then reallocated once its behavior is determined. This has the downside that operators need to reserve a probing node to which every workload has to be migrated for each probe run.

By using RDT cache and memory bandwidth partitioning, we can achieve meaningful probing results on high-interference nodes in a completely online

manner. The configuration section of the prober is shown in Listing 4.17. Using the `implementation` field, we can select a specific `ProbeKernel`, which is an independent process that autonomously performs probing according to its implementation and returns a `SensitivityMetrics` object to the scheduler once it finishes. It includes representative metrics targeted at shared resources: last-level cache usage, memory read and write, prefetching, and the system's kernel, which is also shared among containers.

The sensitivity labels are merely a guideline, not a guarantee that the measured sensitivity fully corresponds to the pressure point. For example, if we measure sensitivity to contention during memory reads, we will also pressure the caches as a side effect. For using sensitivities in optimization, this interaction might not be relevant as long as the overlap is not too large. The microbenchmarks used by the `Default` probing implementation are further introduced in Subsection 4.5.3 and can be seen in Table 4.2. The microbenchmarks are selected and configured to be as distinct as possible with respect to the target contention points, e.g., by using special machine instructions to bypass the cache hierarchy when writing to DRAM.

Each value is clamped to $[0, 1]$ where zero should represent no impact, while one indicates a complete stall if the targeted dimension is under contention. The `isolation` flag enables RDT isolation. The prober will either take a fixed amount of RDT resources and allocate them to the container. This can be configured to be enforced aggressively, meaning by relinquishing other partitions, or to be a best effort. The probe is executed for `default_t` seconds by default, but the scheduler can override the duration. The prober can access the container dataframes to aggregate and derive sensitivities.

The sampling rate during this period can be adapted, by default to 50 ms in between sampling steps. This enables us to run at a low global sampling rate while still performing fast probing, without having to wait for the next dataframe to arrive. Likewise, by sampling multiple times per contention kernel, we can compute averages or eliminate outliers.

```yaml
prober:
    implementation: default
    abortable: false
    isolated: true
    default_t: 12
    warmup_t: 10
    cooldown_t: 0
```

Listing 4.17: Example of a sensitivity probe configuration.

```go
type SensitivityMetrics struct {
    LLC     *float64
    MemRead *float64
    MemWrite *float64
    SysCall *float64
    Prefetch *float64
}
```

Listing 4.18: The `SensitivityMetrics` structure. The methods to create contention for each of these resources or scenarios should be as specific as possible. However, it is impossible to guarantee no interaction. For example, if we aim to trigger the prefetchers, we will inevitably put pressure on the memory bandwidth as a side effect.

Internally, the probe kernel has access to the dataframes, an RDT allocator interface, and a Docker client. This way, a probing implementation can start a deliberate "busy neighbor" that it can move into a common RDT partition with the target container. The `DefaultProbeKernel` executes a series of `stress-ng` [37] microbenchmarks within the neighbor container and measures the impact on the target container by comparing it to a baseline established while the neighbor was idle.

An example of a command targeting the last-level cache is shown in Listing 4.19. The total probing time is shared for determining the baseline and the sensitivities. Hence $segmentTime = \frac{default\_t}{6}$. A full timeline is shown in Figure 4.3.

```go
command:="stress-ng --cache 0 --cache-level 3"

llcIPCE, llcIPC, llcSCP, err := dpk.measureWithWorkload(ctx,
    dockerClient, probingContainerID, containerDF, cores, command,
    segmentTime)

if err == nil && baselineIPC > 0 {
    ipcSensitivity := (baselineIPC - llcIPC) / baselineIPC
    ipcVal := clampSensitivity(ipcSensitivity)
    ipcResult.LLC = &ipcVal
    dpk.logger.WithFields(logrus.Fields{
        "baseline_ipc": baselineIPC,
        "stressed_ipc": llcIPC,
        "sensitivity": ipcVal,
    }).Info("LLC IPC sensitivity calculated")
}
```

Listing 4.19: Example of a microbenchmark launched during probing. The `measureWithWorkload` function executes the specified command within the neighbor container. The decrease in the respective metric (IPC, IPCE, or percentage of stalled cycles) is clamped to $[0, 1]$ and represents the decrease while the command executed for `segmentTime` seconds.

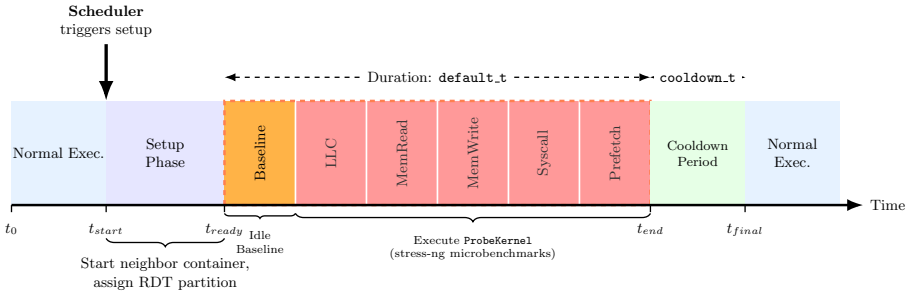


Figure 4.3: After the scheduler issues a probe, the respective probe runner will handle the process end-to-end. The runner will start a new container on the same socket as the target. It will create an RDT partition and add both containers to it. This setup phase usually takes one second if the needed neighbor container image is already present on the host, which is usually the case. For the first `timeSegment` seconds (or milliseconds), the neighbor container will remain idle. The resulting dataframes resemble the container's baseline metrics, since it is isolated from other system noise. After the baseline, the runner will execute a series of commands that target different resources. Upon completion, the runner will return the results to the scheduler. The container cannot be reprobed within `cooldown_t` seconds. The runner will clean up the RDT partitions and restore the former values. The neighbor container is stopped and removed.

4.5.3 Stressor Selection

In this section, we present the synthetic workloads that aim at creating contention on the targeted shared resource. We use stress-ng microbenchmarks, which are designed to fully exploit the hardware and can be configured to further stress the host.

Last-Level Cache Capacity

To isolate Last Level Cache sensitivity, we use the `cache` stressor, which targets level three. The parameter `--cache 0` spawns one worker per logical core, ensuring that the aggregate working set scales with the available processing units. By specifying `--cache-level 3`, `stress-ng` automatically detects the last-level cache size of the system and sets the buffer size to match this capacity. This configuration forces the replacement policy (LRU) to constantly evict lines, resulting in a high volume of capacity misses and starving sensitive victim workloads of cache residency.

Memory Read Bandwidth

We use the `memrate` stressor rather than the standard `stream` benchmark to achieve greater isolation when targeting read sensitivity. Standard stream benchmarks mix read and write operations. By setting `--memrate-wr-mbs 0`, we suppress all write traffic. Conversely, setting `--memrate-rd-mbs 1000000` (an effectively unreachable target) uncaps the read throughput, forcing the stressor to saturate the read data path and maximize queuing delays for any neighbor attempting to fetch data from DRAM.

Memory Write Bandwidth

Generating pure write contention requires bypassing the cache hierarchy to avoid "Read-For-Ownership" (RFO) traffic, where the CPU must fetch a cache line before modifying it. We achieve this using the `--memrate-method write64nt` parameter, which employs non-temporal store instructions, e.g., `MOVNT` on x86. These instructions write directly to the write buffers and subsequently to DRAM, effectively bypassing the caches. Combined with `--memrate-rd-mbs 0`, which disables reads, this creates higher levels of isolated writing.

Kernel Contention via System Calls

While generic syscall stressors measure the mode-switch overhead, true "noisy neighbor" impact in the kernel often stems from lock contention. We utilize `--futex 0` to target the Fast Userspace Mutex system call [38], which is the underlying primitive for thread synchronization. This command generates rapid wait/wake cycles that thrash the kernel's global futex hash table [39]. Contention on these hash table spinlocks introduces non-deterministic latency for any victim workload that employs thread synchronization, providing a realistic proxy for scheduler and lock-based interference.

Hardware Prefetcher Saturation

The aggressor has to saturate the possibly shared hardware prefetchers to starve a sensitive victim. We use the `stream` stressor with `--stream-index 0`, which forces a strictly sequential memory access pattern optimal for engaging the prefetcher logic. Crucially, we append `--stream-madvise hugepage` to utilize Transparent Huge memory pages. This eliminates TLB misses as a bottleneck, allowing the aggressor to run deep enough into the address stream, thereby aiming at denying the victim workload the benefits of automatic hardware prefetching.

Sensitivity	Microbenchmark
LLC	`stress-ng --cache 0 --cache-level 3`
MemRead	`stress-ng --memrate 0 --memrate-rd-mbs 1000000 --memrate-wr-mbs 0 --memrate-bytes 500M`
MemWrite	`stress-ng --memrate 0 --memrate-wr-mbs 1000000 --memrate-rd-mbs 0 --memrate-method write64nt`
Syscall	`stress-ng --futex 0`
Prefetch	`stress-ng --stream 0 --stream-madvise hugepage --stream-index 0`

Table 4.2: The stress-ng microbenchmarks used by the `Default` probing kernel to pressure different resources.

4.6 Scheduler

The scheduler or optimization engine implements policies that aim to achieve different optimization goals, e.g., fairness or noisy neighbor mitigation. To this end, it uses different interfaces to derive decisions and enforce them. The goal is to simplify the scheduler's implementation. To this end, we want to abstract the handling of underlying actions, e.g., moving containers or changing a resource partition. These aspects are wrapped so that the scheduler only has to call a single function. The `Scheduler` interface seen in Listing 4.20 resembles a minimal implementation. The scheduler hence integrates the following aspects and functionalities:

- Move containers across cores, sockets, or nodes (in a multi-node setup)
- Analyze data frames, including RDT metrics
- Bookkeeping about resources
- Call the `Probe` function online to get workload sensitivities
- Allocate RDT resources

```
type Scheduler interface {
    Initialize(accountant *accounting.RDTAccountant, containers
        []ContainerInfo, schedulerConfig *config.SchedulerConfig) error
    ProcessDataFrames(dataframes *dataframe.DataFrames) error
    Shutdown() error
    GetVersion() string
    SetLogLevel(level string) error
    SetHostConfig(hostConfig *host.HostConfig)
    SetCPUAllocator(allocator cpuallocator.Allocator)
    AssignCPUCores(containerIndex int) ([]int, error)
    SetProbe(prober *probe.Probe)
    SetBenchmarkID(benchmarkID int)
}
```

Listing 4.20: The `Scheduler` interface.

The scheduler is initialized with its respective `schedulerConfig`, which can vary depending on the policy. The orchestrator will enqueue new `containers` during the benchmark, which have to be assigned to a core using the `allocator`. The `hostConfig` is the single source for all modules to query host parameters, e.g., the topology, as shown in Listing 4.21.

The architecture of the last-level cache can be queried using RDT and is depicted in Listing 4.22. It is important that all modules use the same global `hostConfig` to avoid discrepancies. The `ProcessDataFrames` is executed periodically during the benchmark and implements the core policy for admitted containers. Listing 4.23 shows a simple example that periodically logs the current cache miss rate of the executed containers. The scheduler and all other components can have individual log levels.

Upon completion of a benchmark, the scheduler will clean up the execution using the `Shutdown` function, which triggers cleanups of the respective modules, e.g., to restore the default RDT partitions. The scheduler is further configured with a `Probe` that it can issue to any container during execution.

```go
type HostConfig struct {
    CPUVendor string
    CPUModel string

    Topology CPUTopology

    L1Cache CacheConfig
    L2Cache CacheConfig
    L3Cache L3CacheConfig
    RDT RDTConfig

    Hostname     string
    OSInfo       string
    KernelVersion string

    logger *logrus.Logger
}

var (
    globalHostConfig *HostConfig
    hostConfigOnce sync.Once
)
```

Listing 4.21: Parts of the host configuration structure.

```go
type L3CacheConfig struct {
    SizeBytes   int64
    SizeKB      float64
    SizeMB      float64
    LineSize    int
    CacheIDs    []uint64
    WaysPerCache int
    BytesPerWay int64
    MaxBitmask  uint64
}
```

Listing 4.22: The last-level cache sub-structure.

```go
func (ds *DefaultScheduler) ProcessDataFrames(dataframes
    *dataframe.DataFrames) error {
    containers := dataframes.GetAllContainers()

    for containerIndex, containerDF := range containers {
        latest := containerDF.GetLatestStep()
        if latest == nil {
            continue
        }

        if latest.Perf != nil && latest.Perf.CacheMissRate != nil {
            ds.schedulerLogger.WithFields(logrus.Fields{
                "container":     containerIndex,
                "cache_miss_rate": *latest.Perf.CacheMissRate,
            }).Debug("Cache miss rate")
        }
    }

    return nil
}
```

Listing 4.23: Example of a simple `ProcessDataFrames` implementation.

4.6.1 Resource Director Technology Integration

The `RDTAccountant` shown in Listing 4.24 handles all interactions and bookkeeping for Resource Director Technology. To this end, it implements a series of carefully synchronized functions that it exposes to the scheduler and probe implementation. Internally, it uses an `RDTAllocator` that wraps RDT function calls, which are prone to errors and must be handled with care. The accountant keeps track of the `SocketState` on the node as well as the user-defined CLOS partitions and the PID mappings to them.

```go
type RDTAccountant struct {
    allocator allocation.RDTAllocator
    hostConfig *host.HostConfig
    logger    *logrus.Logger
    mu        sync.RWMutex

    socket0State SocketState
    socket1State SocketState

    classes      map[string]*ClassAllocation
    containerClass map[int]string
}
```

Listing 4.24: The fields of the `RDTAccountant`.

```go
type SocketState struct {
    TotalWays      int
    AllocatedBitmask uint64
    MemBandwidthUsed float64
}

type ClassAllocation struct {
    ClassName string
    Socket0   *allocation.SocketAllocation
    Socket1   *allocation.SocketAllocation
    Containers []int
}
```

Listing 4.25: Example of the RDT bookkeeping implemented in the `RDTAccountant`.

Beyond the tracking, the accountant is crucial to optimizing the concrete allocation decisions requested by the scheduler. To avoid fragmentation of RDT resources, the accountant has to scan the current configuration to determine the current best fit, e.g., using the `findBestFitBitmaskForWays` seen in appendix Listing A.4, which aims at finding the best possible CLOS bitmask for a given request of cache ways for a socket.

This problem, however, breaks down into several optimization decisions and trade-offs. For example, if a request cannot be served in its current state because there are not enough contiguous bits, one could reallocate other partitions to defragment the overall allocation. While this could enable more requested partitions, it potentially has the cost of invalidating the caches of unrelated classes. Note that after reallocation, the lines in the former ways remain valid until a new partition holder evicts them. Similarly, the accountant has to manage pressure transfer. For example, when a container is underallocated in terms of cache ways, it will likely start using more bandwidth.

The allocator interface seen in Listing 4.26 shows all the higher-level functions it exposes to the accountant. Internally, the allocator has to manage the technical integration of RDT using the respective library function [31]. The interactions with RDT have to be synchronized using the `rdtguard` to avoid race conditions with collectors, which also interact with RDT to create monitoring groups and scrape the current allocation independently. The RDT allocations cannot simply be created or updated individually. We rather have to keep track of our current partitions and reapply them on every change using the utility function `applyManagedConfigLocked` function seen in appendix Listing A.5, which builds the `configClasses` map of managed allocations and the `system/default` partition.

The full allocation is then applied using `rdt.SetConfig(config, force)`. Hence, the `CreateRDTClass`, `UpdateRDTClass`, and `DeleteRDTClass` only update the `managedConfigs` map, which is applied by `applyManagedConfigLocked`. Upon a class deletion, we have to flush the containers that are part of it, as RDT cannot release the respective PIDs automatically and refuses to delete a non-

empty CLOS. However, we can move containers between classes freely, e.g., using the `AssignContainerToClass` function of the allocator. Upon benchmark shutdown, the managed classes are consolidated in the `Cleanup` function, which safely removes all classes and restores the `system/default` class if it was altered.

```
type RDTAllocator interface {
    Initialize() error
    CreateRDTClass(className string, socket0, socket1 *SocketAllocation)
        error
    UpdateRDTClass(className string, socket0, socket1 *SocketAllocation)
        error
    CreateAllRDTClasses(classes map[string]struct {
        Socket0 *SocketAllocation
        Socket1 *SocketAllocation
    }) error
    AssignContainerToClass(pid int, className string) error
    RemoveContainerFromClass(pid int) error
    GetContainerClass(pid int) (string, error)
    ListAvailableClasses() []string
    DeleteRDTClass(className string) error
    Cleanup() error
}
```

Listing 4.26: The `RDTAllocator` interface.

4.7 Visualization, Analysis & Reproducibility

The experimental setup serves as a test ground for quantifying the interference of different workloads. Especially during the implementation phase of a schedule, it becomes increasingly hard to follow the decision-making and identify emerging interference patterns.

To facilitate the overall analysis and debugging of schedulers, we added a comprehensive dashboard [40] which is connected to the benchmarking database. Hence, the results are immediately available after a benchmark run is completed. Besides visualizing the evolution of the metrics per container, the dashboard aggregates metrics such as the global cache miss rate or stalled cycles, e.g., to compare different schedulers. The dashboard also displays the configuration and metadata used for the experiment, e.g., the host topology and the container-core mapping over time. As mentioned, the benchmark results are written to the database *after* the completion to avoid unnecessary interference with the experiment. Essentially, the data remain in the DRAM of the testbed after it was collected unless the scheduler explicitly requests it or it was accidentally prefetched by the hardware.

4.7.1 Examples

In this section, we show a few examples of the visualizations. In the example benchmark, the containers start at different times and on different sockets. They execute different workloads such as installing packages (`install`), compressing a large file (`compression`), copying a tiny buffer (`copy`), dual-threaded operations on a 3D matrix (`matrix-3d`), and looking up a random buffer of 12 MB (`neighbor` and `neighbor-2`). The container `neighbor-2` is the only one running on socket one.

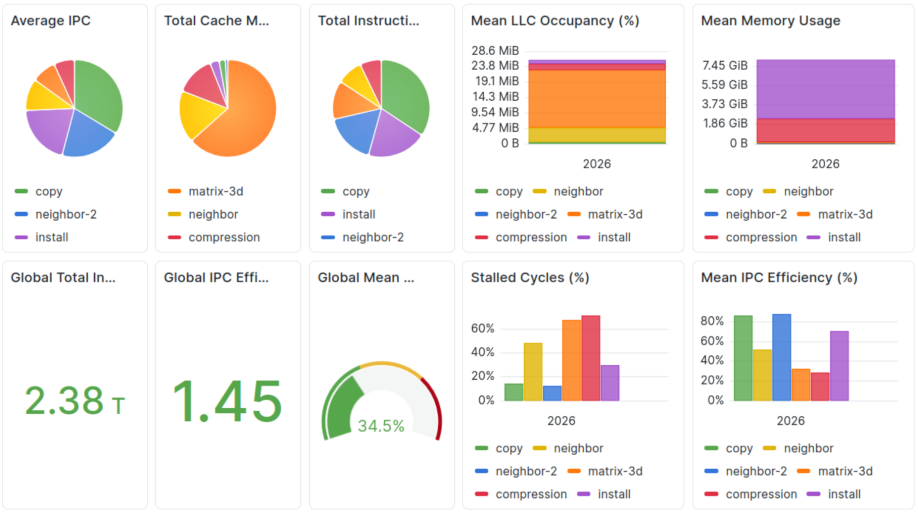


Figure 4.4: Parts of the overview panels which display aggregate metrics such as the total executed instructions of all containers, the mean instructions per cycle, the mean cache miss rate, the mean last-level cache occupancy, or the percentage of cycles per container where the pipeline stalled. For example, the `neighbor` container has significantly more stalled cycles because it has to share resources with the others, unlike `neighbor-2`, which runs the same workload but on socket one.

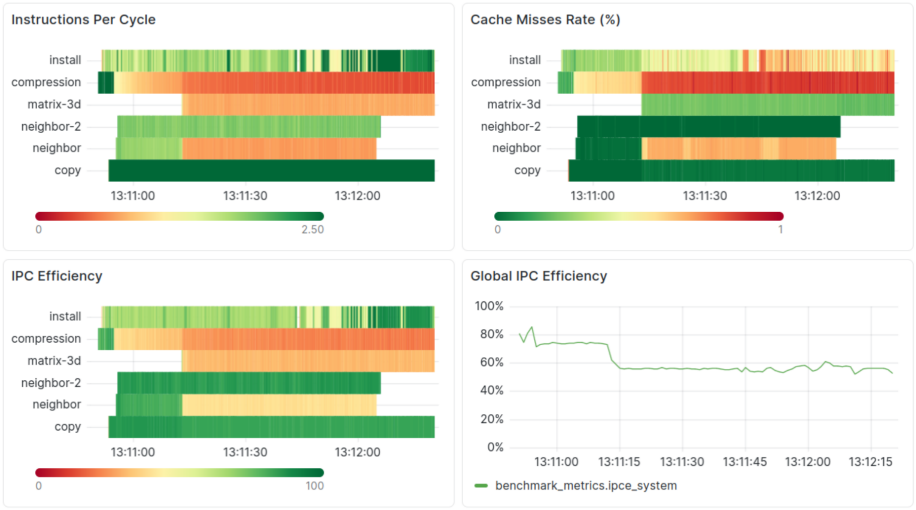


Figure 4.5: Heatmap panels displaying different metrics and the global IPC efficiency. There is a clear drop in global IPC efficiency once the resource-intensive `matrix-3d` container is admitted. For example, at this point `neighbor` and `compression` start to have cache misses.

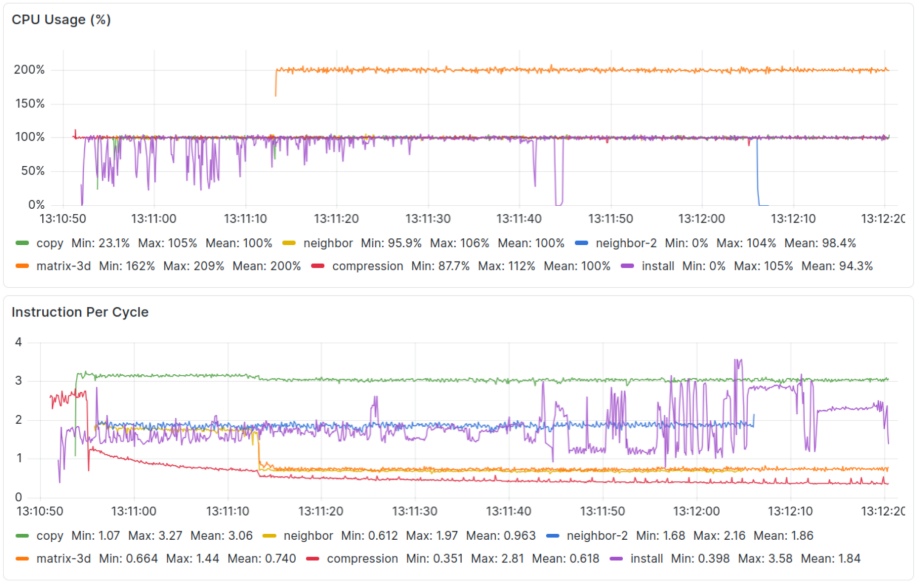


Figure 4.6: CPU Usage and IPC panels per container. All containers run on a single core except `matrix-3d`, which uses two. The IPC of `neighbor` drastically drops once `matrix-3d` starts executing. However, when `neighbor` finishes its execution, the IPC of `matrix-3d` remains the same, which indicates that it was the dominant neighbor. While most containers use their cores consistently, the `install` container also has periods when it underutilizes its cores, likely because it is waiting for slow I/O. The IPC of the `compression` container slowly decreases, even before `matrix-3d` is started.

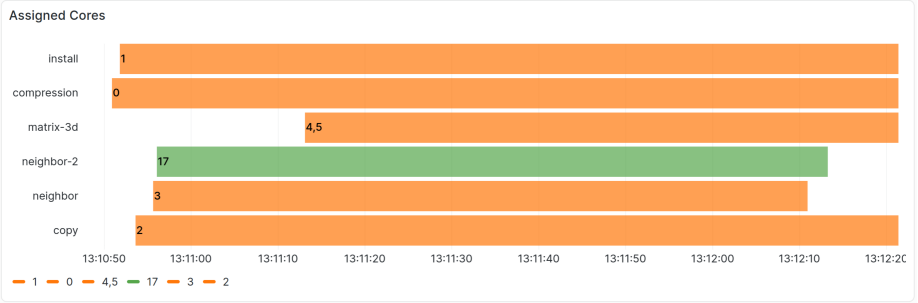


Figure 4.7: The scheduler assigns each container its requested number of cores, which are tracked independently by a collector.

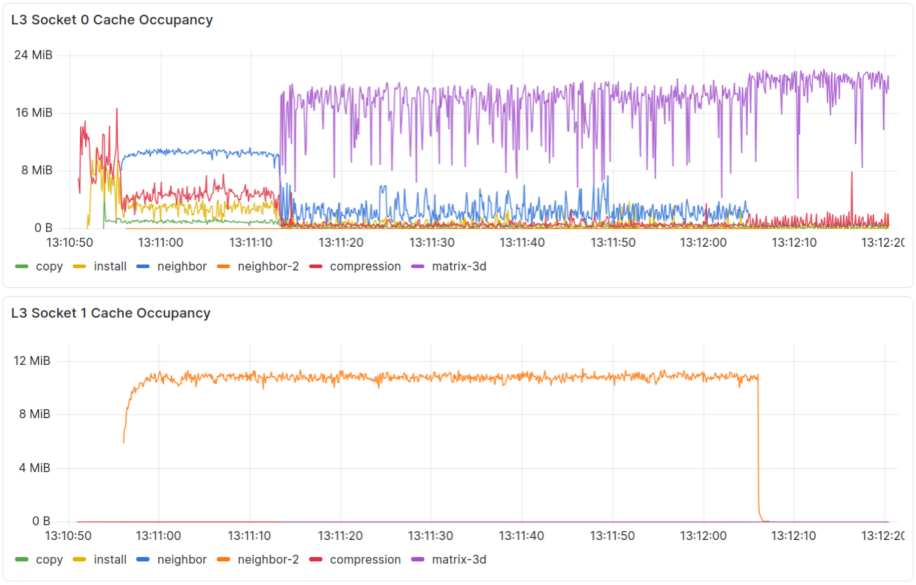


Figure 4.8: The last-level cache occupancies of the containers on each socket. Once the two processes in `matrix-3d` start executing, they basically occupy the 24 MB large last-level cache, "pushing" the other containers on socket zero out.

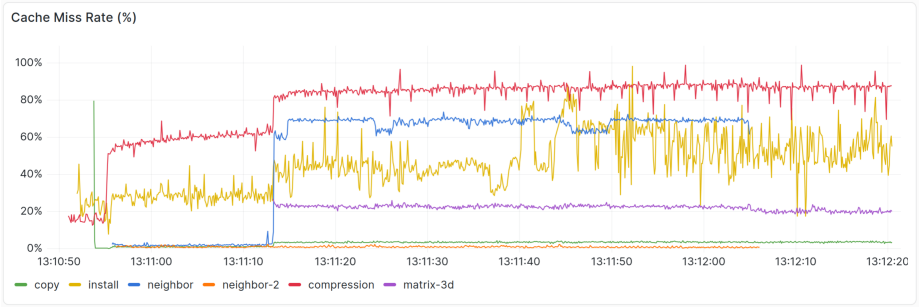


Figure 4.9: The overall cache miss rate per container. All containers on socket zero start to experience more cache misses once `matrix-3d` starts executing.

Figure 4.10: The percentage of stalls and how many of them include last-level cache stalls. Overall, the `compression` container is slowly stalling more over time. However, the percentage of stalled cycles caused by misses in the last-level cache remains steady apart from the moment `matrix-3d` starts executing. This indicates that the source of stalls is not memory, and hence is shared-resource-related.

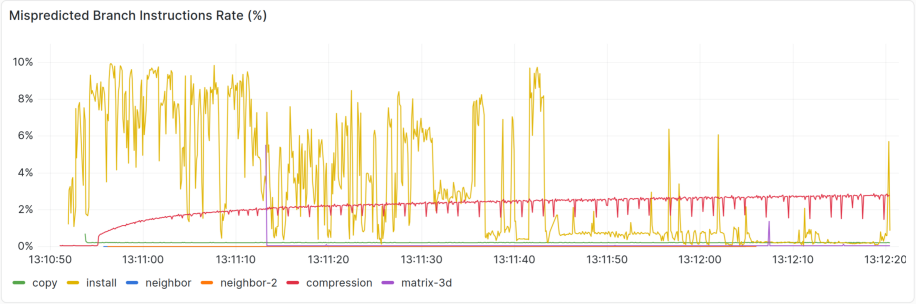


Figure 4.11: The branch misprediction rate panel. After an initialization phase, the `compression` container suffers from more and more branch mispredictions, which correlate with its percentage of stalled cycles, which follow the same curve. The `install` container undergoes many execution phases, while its branching behavior is easy to predict.

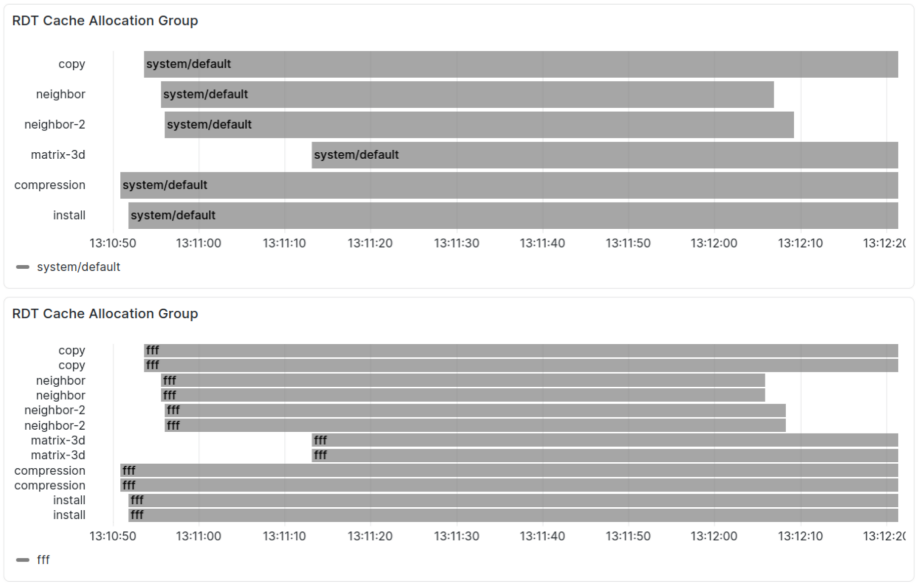


Figure 4.12: The RDT class the containers where part of. In this benchmark, the scheduler issued no RDT class allocation. Hence, all the containers were admitted and remained in the `system/default` group. The bitmask (`fff`) corresponds to 111111111111_2, allowing the group to use all 12 cache ways on both sockets.

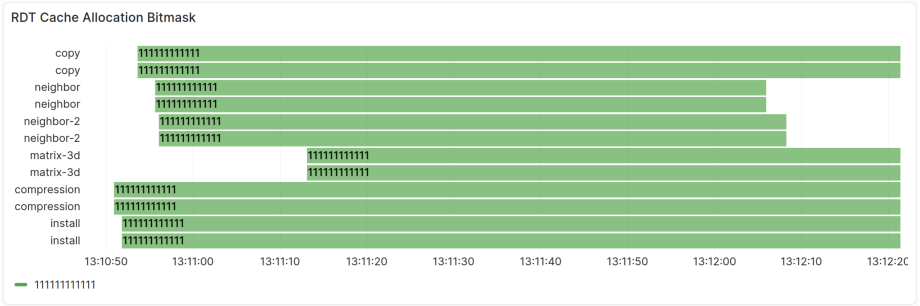


Figure 4.13: The specific cache-way bitmask for each container and socket, which helps to determine which cache ways were allocated, or for example, which groups share certain ways.

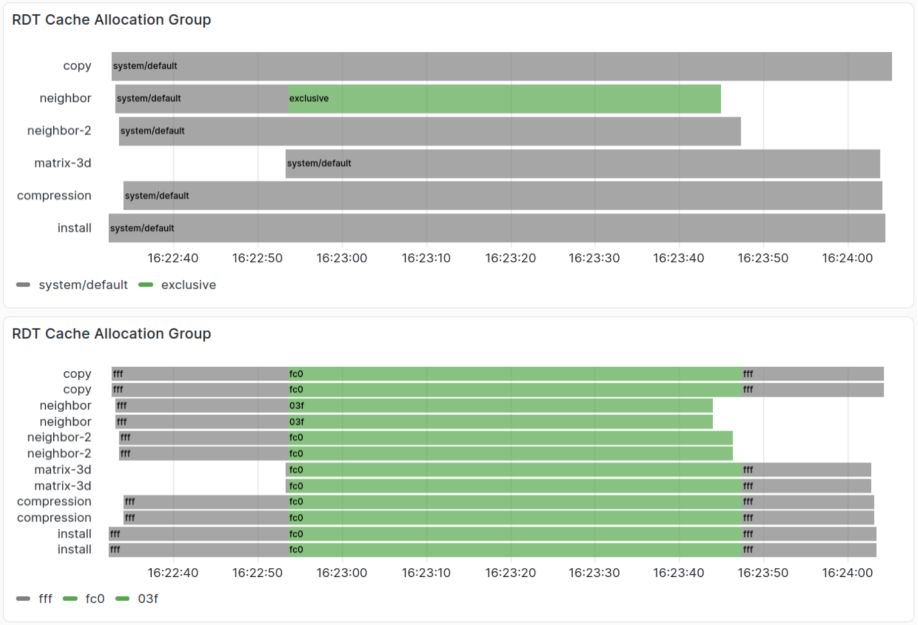


Figure 4.14: In this example, the scheduler allocated six fully (meaning both sockets) exclusive cache ways to the `neighbor` container upon the admission of `matrix-3d`. After `neighbor` finishes, the default is restored. During the exclusive allocation, the `system/default` group uses the cache ways 6–11, which corresponds to the $fc0_{16}$ or 111111000000_2 bitmask. The exclusive group has access to the cache ways 0–5 with the bitmask $03f_{16}$ or 000000111111_2.

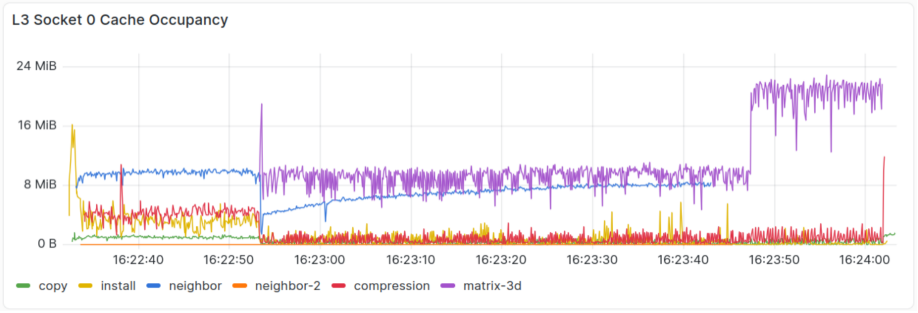


Figure 4.15: As a result of the exclusive allocation, the `neighbor` can claim back some of the last-level cache while `matrix-3d` is throttled. However, pressure can transfer to the memory bandwidth, which in turn throttles `neighbor` as it heats up its cache partition. Once `neighbor` finishes, the other containers can use the resources again.

Chapter 5

Experimental Setup

In this chapter, we present the testbed and the workloads used during development and experimentation. The experimental setup is compatible with all modern Intel processors. Support for the Intel Resource Director Technology integration is contingent on a concrete model. A selection of compatible CPUs can be seen in Table 2.1. We distinguish between batch and priority jobs, each with a target metric to be met. The jobs are generated from a pool of workloads, which are introduced in the following sections.

5.1 Load Scenarios

In this section, we present the load scenarios used to evaluate the scheduling policies. The interarrival times and lengths of jobs follow a normal distribution. We select fixed parameters of these distributions for all scenarios such that the host will always be fully utilized. For a host with 32 physical cores, the mean interarrival time is defined as $\mu = 5$ with a standard deviation of $\sigma = 2$, where jobs request 1.5 cores on average. In combination with a job-length distribution with $\mu = 140$, $\sigma = 30$, and min $= 30$, this will slowly increase the queue length. Since the jobs are moderately long, we can test the scheduler's ability to respond to frequent changes in jobs. In particular, this emphasizes the scheduler's bookkeeping, which is responsible for identifying optimizations and efficient resource allocation and consolidation.

The jobs are categorized according to their expected sensitivity to interference. We consider 75% of the overall jobs to be moderately sensitive and 25% to be highly sensitive to interference. The defining parameter that varies across the following load scenarios is the fraction of jobs with a performance target assigned and therefore flagged as priority. The resource demands of the priority jobs vary. Therefore, the scheduler will need to employ allocation probing to determine the appropriate allocation. The respective scenarios are described in the following Sections 5.1.1, 5.1.2, and 5.1.3 and are summarized in Table 5.1.

In all cases, we do not wait for the queue and execution to drain. After

`max_t`, which is set to one hour, the benchmark will be stopped. For probing, we use a minimum warm-up time of 3 seconds and a minimum cool-down time of 1 second, with a maximum probing duration of 12 seconds. The balancing and swap option of the dynamic scheduler allows it to consolidate and move batch containers to enable admissions or probings. Hence, we enable it for the presented benchmarks to allow more movement.

Scenario	**Priority Jobs**	**Resource Demand**
Low Demand	10%	$\ll$ Capacity
High Utilization	20%	$\approx$ Capacity
Exceeding Demand	50%	$\gg$ Capacity

Table 5.1: Summary of the load scenarios. The distributions of job interarrival time and length are fixed to ensure full utilization of the host's compute cores. The fraction of priority jobs varies, creating different demand for RDT resources.

5.1.1 High Utilization

In this scenario, 20% of the arriving jobs will be selected from the pool of priority jobs, and the scheduler will assume that there is an allocation combination that allows the container to perform much better. The specific fraction of 20% will statistically yield a resource demand at or slightly below the amount of RDT resource the host can provide. For example, of the 22 allocatable cache ways, 19–21 will be exclusively allocated to containers at any given time, while the remaining ones are used as additional resources for non-priority jobs.

5.1.2 Exceeding Demand

In this experiment, 50% of the arriving jobs will be priority jobs. This will cause more demand for shared resources than the host system can provide, e.g., the demand for exclusive memory bandwidth will exceed 100%. Hence, some priority jobs will never receive their full allocation, or may even finish without any attempt by the scheduler to allocate them.

5.1.3 Low Demand

In this scenario, 10% of the jobs will be flagged as priority. As a result, it is likely that every admitted priority container can be immediately probed and allocated with its resource demand. Further, the scheduler has time to reprobe containers, e.g., if they fell under their $IPCE^{target}$ after changing an execution phase. Likewise, more resources will be available to the general pool, which will be exclusively used by non-priority containers.

5.2 Workloads

One challenge of interference- and priority-aware scheduling is the combination of non-linear resource demands and changing application configurations. This means that slight differences in the configuration of the same, previously profiled application can fundamentally change its resource demand and interference sensitivity. When changing the input data size, this typically involves a loss of locality due to the cache's capacity. To address this, we define multiple configurations for all workloads.

5.2.1 PostgreSQL

PostgreSQL [41] is a widely adopted open-source database system. Its benchmarking tool, `pgbench`, can be used to generate load. Since we are measuring the effects of shared resources, we offload the load generator to a secondary server to avoid interference. To evaluate different characteristics, we benchmark a single-core and a four-core setup.

We distinguish the working set size using the `-s` parameter of the `pgbench` configuration, which determines a scaling factor. For the `small` variant, we set $s = 1$, and for the `large` variant, we set $s = 30$. For database operations, we perform a 50% split between read and write operations. Additionally, we run read-only benchmarks with the `-S` (*select*) option and write-only benchmarks with the `-N` (*no-vacuum*) option. PostgreSQL supports 1 GB Huge Page, which we employ in all benchmarks.

5.2.2 Compilation

We use GCC [42] to compile the Linux kernels as an example for compiling a large project. Typically, building and compiling a large project undergoes different phases, which challenge an interference-aware scheduler. We use the standard Makefile. The benchmarks include different core-counts to test the behavior of concurrent builds, e.g., `(make -j8)`.

5.2.3 Compression

The 7-Zip [43] compression tool and library implement compression algorithms that aim to achieve higher compression ratios. The levels range from 1 (lowest) – 9 (highest) and are configured via the `-mx` parameter. The implementation of each level can significantly alter 7-Zip's resource usage patterns and demand; therefore, it is a useful benchmark for an interference-aware scheduler, as applications can exhibit diverse behavior.

Additionally, we run the application with different amounts of worker threads using the `-mmt` option. For the input data, generate a per-container, randomly sampled data blob of 1 GB. In this way, we can exclude cache synergy between unrelated containers.

5.2.4 FFmpeg

FFmpeg [44] is a feature-rich video library and tool. Similar to compression, it bundles different algorithms, which can lead to significantly different resource demands. Additionally, the implementations of FFmpeg are oftentimes highly optimized, e.g., using SIMD or parallelization. Hence, the jobs tend to exploit their resources to a high and constant extent. As input, we use large video files, which are exclusive to each job; we exclude cache synergy. The methods we test are `blur`, `scale` & `hue`, which applies a filter, and high compression using `-c:v` with `veryslow` presets. The full commands can be seen in Appendix Listing A.6. Further, we test the application with different amounts of available compute cores.

5.2.5 stress-ng Microbenchmarks

The stress-ng [37] tool bundles a large amount of microbenchmarks which cover all aspects of computing. In addition to typical parameters such as work size or thread count, the tool enables fine-grained configuration of various aspects. This includes considerations of the host, such as the total cache size or per-cache-line size.

Furthermore, many benchmarks offer additional configurations, e.g., to use specialized machine instructions to increase or decrease pressure on a particular resource. One example is non-temporal writes, which avoid filling the cache while still using memory bandwidth. We use a variety of microbenchmarks, including `tsearch`, `matrix3d`, and `stream`. The `cpu` benchmark uses sub-methods such as `bitops`,`sieve`, or `gcd`. The tool provides multiple benchmarks that cover all aspects of computing, making them relevant to an interference-aware scheduler.

5.2.6 XGBoost Training

XGBoost [45] is a distributed gradient boosting library optimized for distributed computing. In the project, we use binary classification training on synthetic data. This method is typically CPU-intensive and highly parallelizable, which is why we test the workload with different numbers of workers.

5.2.7 Neighbor Microbenchmark

The neighbor microbenchmark is a low-level memory stress tool designed to allocate a specified buffer size. Further, it enables configurable backing memory types, such as standard 4 KB pages, Transparent Huge Pages (THP, 2 MB), or explicit HugeTLB (1GB). Computationally, the microbenchmark consists of looking up a fixed-sized buffer in different patterns. These include sequential, strided, and random. The relevant parts of the implementation can be seen in Appendix Listing A.7.

After the setup and initialization, the program enters a Read-Modify-Write loop. The pressure of the `random` configuration is further increased by issuing

multiple operations in a tight inner loop. Since the modified entries are never used, the `volatile` keyword is used to prevent compiler optimizations that could reduce pressure. The neighbor program is primarily used as a priority job, especially to study the impact of different page sizes on allocation probing.

5.3 Testbed

The results of this project were developed and tested on different host systems. A list of testbeds can be seen in Table 5.2. The testbeds cover different hardware generations. Hence, the available RDT features vary.

In the final evaluation, we use a dual-socket setup with the Intel Xeon Silver 4314 seen in Figure 5.1. It features two 16-core CPUs with 24 MB, 12-way last-level cache. The Intel Xeon Platinum 8360Y seen in Figure 5.2 features 36 cores, which share a 54 MB last-level cache. The testbed could be useful for future development, as it divides cores into high- and low-priority categories. The Intel Xeon Gold 5512U seen in Figure 5.3 (52.5 MB, 28 cores) uses a 15-way cache. This can be beneficial, as the allocation granularity is finer. In this project, we do not use explicit hyperthreading. Each container is mapped to distinct physical cores.

Testbed	**Processor**	**Sockets**
CloudLab `sm110p` Node	Xeon Silver 4314	Single
CloudLab `sm220u` Node	Xeon Silver 4314	Dual
CloudLab `r650` Node	Xeon Platinum 8360Y	Dual
CloudLab `d750` Node	Xeon Gold 6326	Dual
Mare Nostrum 5 [46]	Xeon Platinum 8480+	Dual
CloudLab `c6620` Node	Xeon Gold 5512U	Single
CloudLab `d760p` Node	Xeon Gold 6526Y	Single
CloudLab `d760` Node	Xeon Gold 6548Y+	Single
CloudLab `d760-gpu` Node	Xeon Gold 6548N	Dual
CloudLab `d760-hgpu` Node	Xeon Gold 6548N	Dual
CloudLab `d760-hbm` Node	Xeon Max 9462	Dual

Table 5.2: Selection of testbeds with Intel Resource Director Supported CPUs.

5.3.1 Intel Xeon Silver 4314

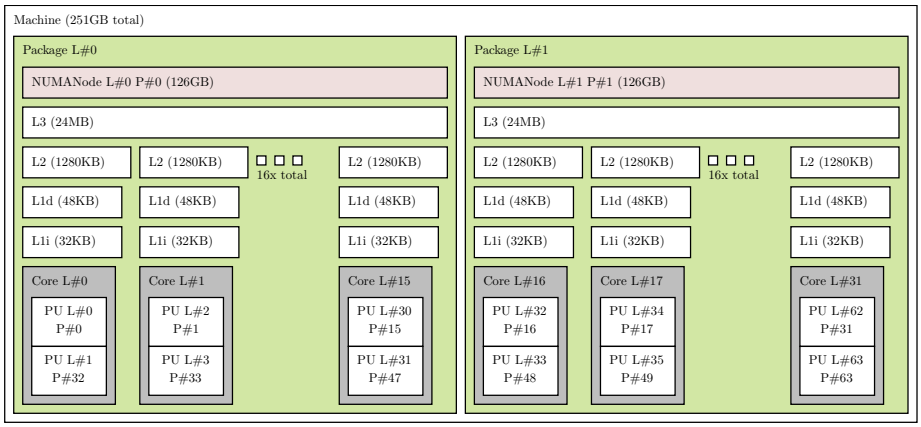


Figure 5.1: The topology of a dual socket Xeon Silver 4314 processor system installed in the `sm220u` machine type [47] at CloudLab, Wisconsin [48].

5.3.2 Intel Xeon Platinum 8360Y

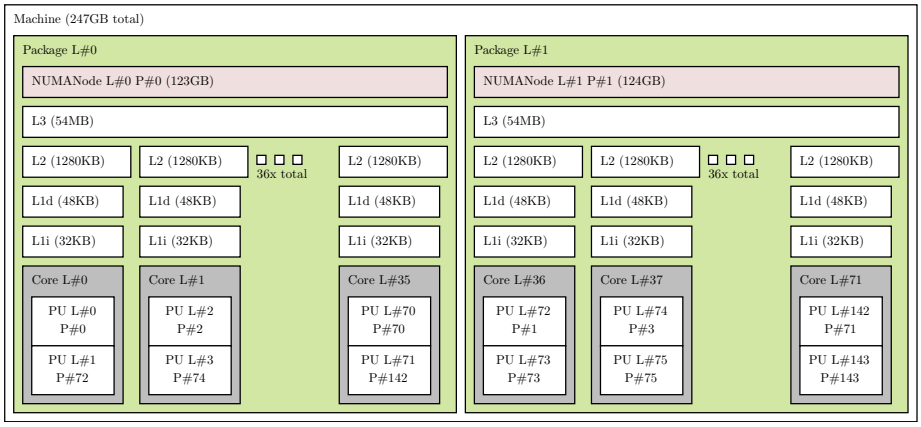


Figure 5.2: The topology of dual socket system with Intel Xeon Platinum 8360Y installed in the `r650` machine type [47] at CloudLab, Clemson [49].

5.3.3 Intel Xeon Gold 5512U

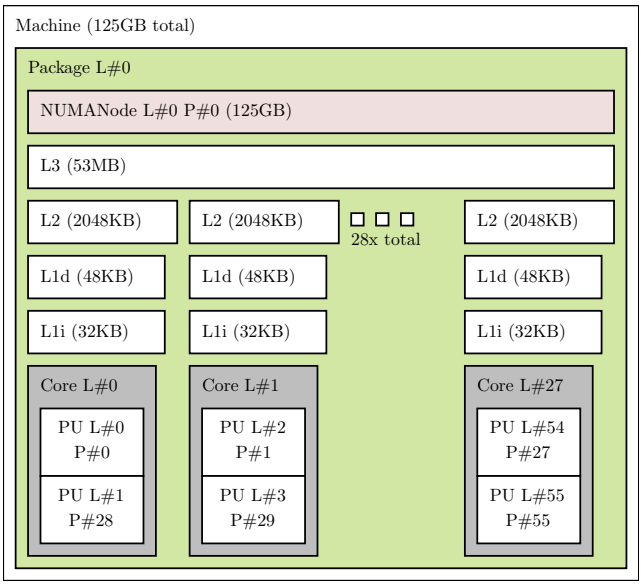


Figure 5.3: The single socket topology of the `c6620` machine type [47] at CloudLab, Utah [50] with a Intel Xeon Gold 5512U processor.

Chapter 6

Validation

In this chapter, we will validate the experimental setup that was developed and used in this project. This includes an analysis of the Intel Resource Directory capabilities. We demonstrate how last-level cache slicing in combination with RDT partitioning can affect experimental results. Specifically, we show how partitioning can incur significantly more conflict cache misses depending on the memory page sizes used. Further, we experimentally demonstrate that RDT enables sensitivity probing in noisy environments. Yet, we evaluate the potential bias it can introduce.

6.1 Last-Level Cache Partitioning with RDT

We encounter a significant performance anomaly when containers use standard four-kilobyte or transparent two-megabyte memory pages. We observe a drastically higher rate of misses compared to identical workloads backed by one-gigabyte huge pages, particularly when the cache allocation is restricted via RDT. This phenomenon was briefly introduced by Sohal et al. [51]. It undermines the deterministic guarantees that RDT is designed to provide, particularly when the container is allocated a partition equal to or greater than its working set size. In this section, we analyze this discrepancy specifically targeting the Intel Xeon Silver 4314 processor [14]. We analyze how an undocumented hash function for cache slicing increases conflict misses and self-eviction when using normal page sizes. Further, we show how this effect is amplified when reducing the number of cache ways accessible to a process by applying an RDT cache partition.

6.1.1 Cache Slicing & Pages Sizes

The last-level cache of modern Intel hardware is divided into slices [52]. The generation of processors used in our testbed typically features one slice per physical core. Using a custom tool [53], we find that the Intel Xeon Silver 4314

indeed features sixteen Caching Home Agents (CHA), which strongly indicates one slice per core. The tool uses a Load-Flush-Measure loop to identify which CHA is active for a load to a given address. The results of the experiment are shown in Appendix B.1.

Mapping physical addresses to these slices is subject to an undocumented hash function [54]. For the specific case of the Ice Lake Xeon Gold 6330 (28 cores/slices), the parameters of this hash function were reverse-engineered [55]. The hash function of the Xeon Gold 6330 uses bits 6 to 37 of the physical address. The bits 6 through 19 index into a long sequence of 2^{14} elements, and bits 20 through 37 are used to select a permutation. However, the different slice count of our system most likely alters the specific bits used by the hash function. This processor is a sibling of our testbed. It shares the same mesh interconnect, and its specific hash function is likely conceptually related to the one used in our testbed. Hence, we will use the conceptual 33-bit lower layout shown in Figure 6.1 to analyze the interaction among page sizes, the hash function, and the corresponding last-level slice mapping.

Four-kilobyte pages are the default, two-megabyte transparent pages have to be enabled manually, and huge page tables of one gigabyte have to be explicitly reserved and requested by an application. The bits the hardware uses to determine the destination slice are independent of the respective page size. However, the relative location in physical memory of the pages is subject to different probabilities depending on their size. The hash function aims for the following properties:

Let S be the number of slices and PA a physical address. Let $\mathrm{PA}_{\mathrm{next}}$ represent the physical address of the subsequent cache line in a sequential stream.

Sequential Interleaving

To ensure that contiguous streams utilize all cache slices, the mapping of the current line must differ from the mapping of the next line:

$$\mathrm{Hash}\big(\mathrm{PA}[32{:}6]\big) \;\neq\; \mathrm{Hash}\big(\mathrm{PA}_{\mathrm{next}}[32{:}6]\big) \tag{6.1}$$

Uniform Distribution

For random access patterns, the hash function distributes addresses evenly across all slices. For any specific slice index k:

$$\forall k \in \{0, \ldots, S-1\}, \quad \mathbb{P}\Big(\mathrm{Hash}\big(\mathrm{PA}[32{:}6]\big) = k\Big) = \frac{1}{S} \tag{6.2}$$

The implementation of the hash function is optimized for data-intensive applications, which may use larger pages. For one-gigabyte pages, most of the hash bits [32:6] overlap with the `Page offset` part of the address, which is the same as in virtual memory and hence contiguous. Sequential access within a page guarantees the former sequential interleaving. For random access, we still know that the data within a page will be evenly distributed across slices. For

standard, transparent pages, however, this continuity is not provided in most cases.

Furthermore, it is much more likely that the frames belonging to the smaller pages are scattered across physical memory as the kernel has more opportunities to fill in fragmentation. Especially in the case of four-kilobyte pages, this decouples the virtual continuity from the physical reality used to compute the hash function. Hence, contiguous memory accesses can cause conflicts when their physical addresses map to the same slice, putting pressure on one slice while others remain underutilized. Random accesses are subject to potentially uneven distribution of mappings, as can be seen in Figures 6.2 and 6.3.

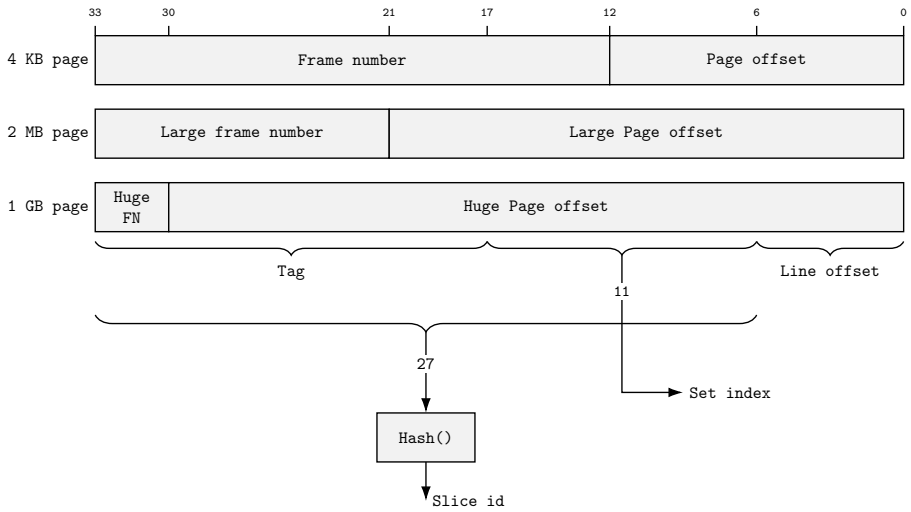


Figure 6.1: Exemplary bit mappings for different choices of page sizes. The address and hence the tag can extend up to 52 bits in Ice Lake.

6.1.2 Unused Cache & Conflict Misses

Figures 6.2 and 6.3 show the cache miss rates and last-level cache occupancies of separate benchmark runs in which we look up a random index of a 6 MB buffer on a 24 MB 12-way last-level cache Intel Xeon Silver 4314 processor. We run separate experiments in which containers use standard four-kilobyte pages, two-megabyte transparent huge pages, and one-gigabyte huge pages, which must be manually requested by the application. We run each configuration with a fixed, exclusive last-level cache partition of 11, 6, 4, and 3 cache ways. The remaining cache way is used by the benchmark driver and other host processes.

The problem manifests as processes backed by standard or huge transparent pages experience orders of magnitude more cache misses when running on

smaller partitions. This is unexpected, as the 4-way and 3-way partitions should still sum to a size large enough to hold the 6 MB buffer. However, only the results of the containers using one-gigabyte pages follow this intuition. The spike in cache misses corresponds to the buffer initialization. Depending on how many lines were evicted between initialization and the start of the hot loop, there is a brief period during which the remaining lines are fetched again. The one-gigabyte page containers with larger partitions occupy a much higher percentage of it, which is most likely due to aggressive prefetching of unrelated lines that never get evicted. Afterwards, the cache miss rate remains steady for all containers.

Since the container itself is the only significant process running in its large enough partition, we can conclude that the majority of these misses are conflict misses. This means there are too many indices in the buffer that are mapped to the same sets, causing constant self-eviction. With smaller page sizes, the containers struggle to fill smaller cache partitions, such as the three- or four-way partitions, which amplifies the previously described effect, as shown in Figure 6.4.

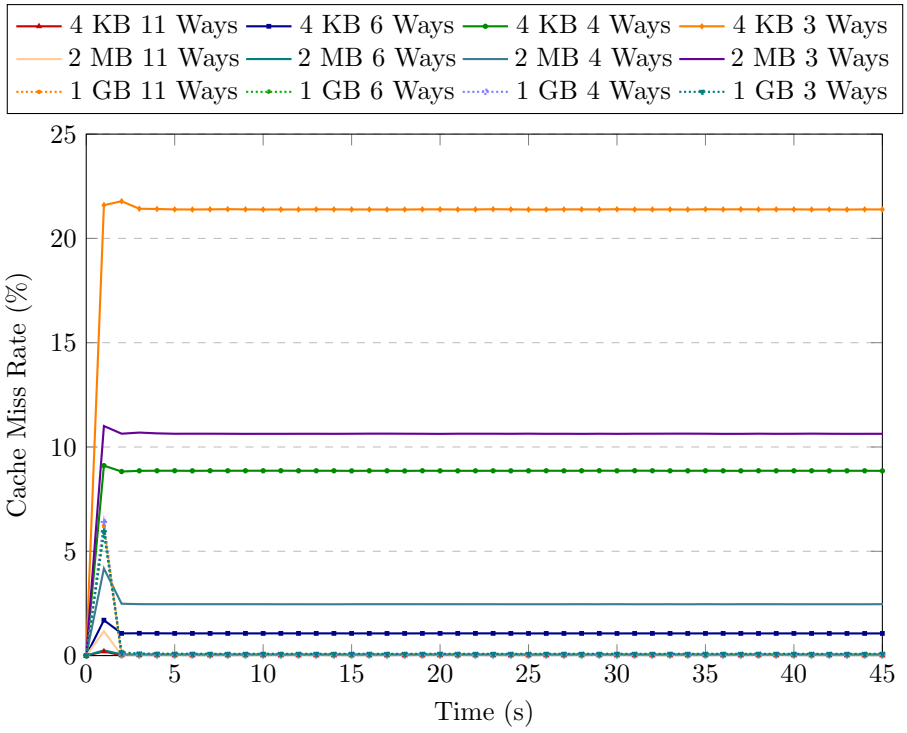


Figure 6.2: The Cache Miss Rate (%) per container for different page sizes and way allocations.

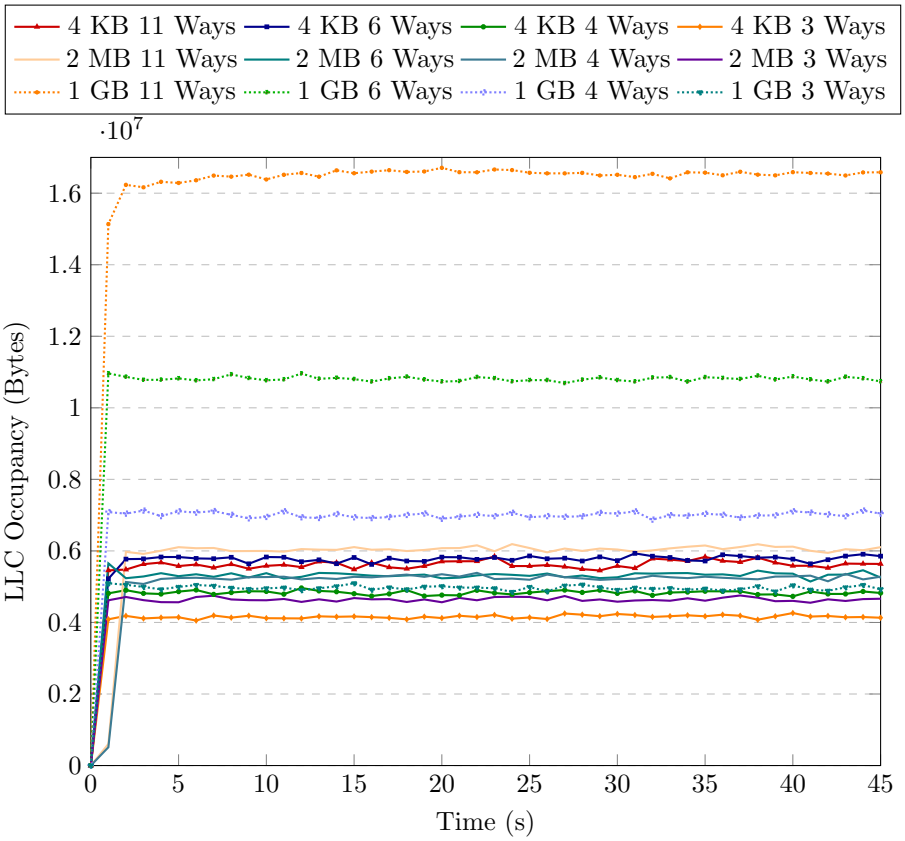


Figure 6.3: The LLC Cache Occupancy (Bytes) per container using different page sizes. On this 12-way, 24 MB LLC machine, the sum of each way is 2 MB. The memory was explicitly defragmented after each run.

6.1.3 Shareable Partitions

The partition units of RDT are rather coarse, typically per-way, with a minimum of one. If we enable the `shareable` mode for RDT, we can interleave bitmasks of different partitions such that certain cache ways remain shared. For example, this could be useful if processes only use the full partition periodically. If they are part of an exclusive partition, they will remain unused, even if others would have demanded. However, we encounter unpredictable performance patterns

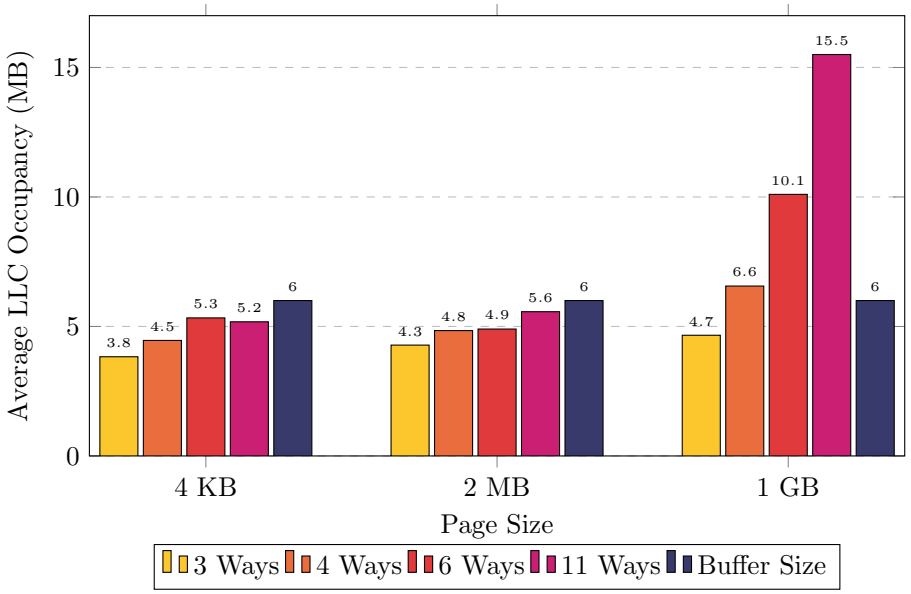


Figure 6.4: The average last-level occupancy (MB) per container using different page sizes in reference to the buffer of six megabytes.

when interleaving cache partitions. In the following example, we run the two containers `c0` and `c1`, which allocate and randomly look up a 10 megabyte buffer.

The containers use two-megabyte pages. Figure 6.5 shows the base case where we do not allocate any resource, hence the containers and the system are sharing. The results from Figure 6.6 show a configuration where the containers share a ten-way partition. Similar to the experiments seen in Section 6.1.2, this leads to an increase in conflict misses due to the reduced associativity of the available last-level cache. The cache misses further increase when we run the containers in exclusive 5-way partitions, as shown in Figure 6.7.

In the benchmark depicted in Figure 6.8, each container has exclusive access to four ways and an additional two ways, which are shared. Although container `c1` shows a slightly lower miss rate than in previous experiments with partitioning, `c0` appears to be heavily penalized, with an average cache miss rate of 13.3%. This creates an average gap of 9.37 percentage points of the miss rates between `c0` and `c1`. In this example, the bitmasks used for the allocation are defined as follows:

$$\texttt{c0} = 000000111111_2$$

$$\texttt{c1} = 001111110000_2$$

and the two upper ways are used for the `system/default` group. Hence, container `c0` has to share its upper bits while the bits that are interleaving for

`c1` are the lower ones. Reversing this assignment effectively reverses the penalty, as can be seen in Figure 6.9. This phenomenon is fully reproducible.

The difference in cache miss rate is amplified to an average gap of 14 percentage points when four ways are shared, and three are exclusive, as shown in Figure 6.10. Sharing six cache ways slightly decreases the gap to an average of 10.37%, as can be seen in Figure 6.11. This phenomenon indicates that, depending on the location in the bitmask, ways in certain slices are less likely to be filled by processes using 2 MB pages.

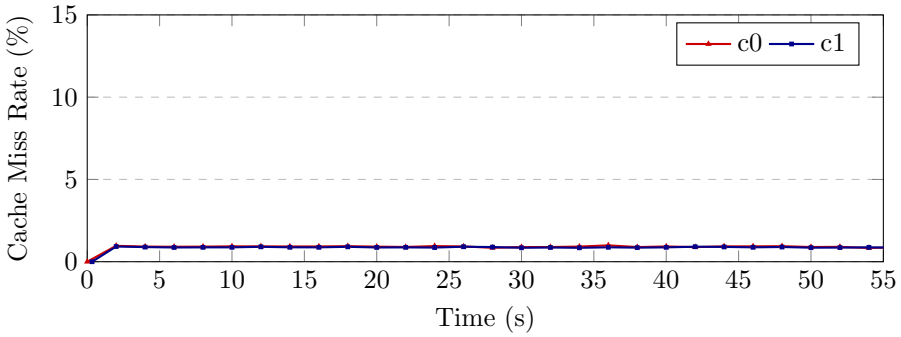


Figure 6.5: Miss rates of the concurrently running containers sharing all ways among each other and with the system. The containers are backed with 2 MB pages.

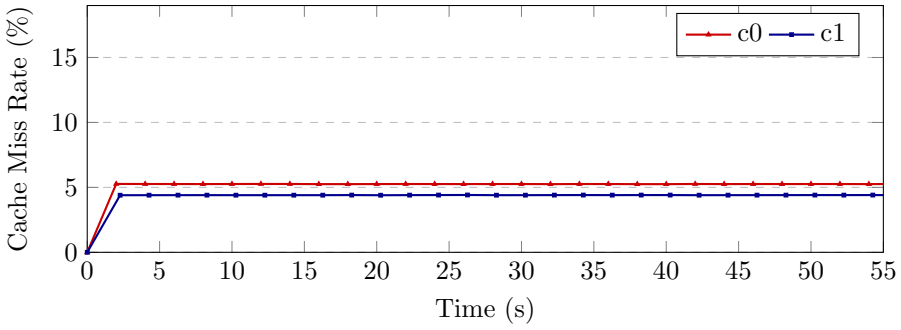


Figure 6.6: The cache misses when both containers run isolated from the system inside a ten-way partition, which is fully shared.

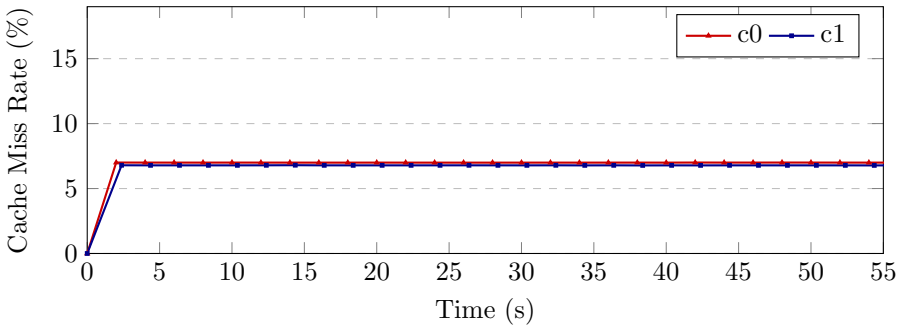


Figure 6.7: The cache miss rate when each container runs in an exclusive five-way partition.

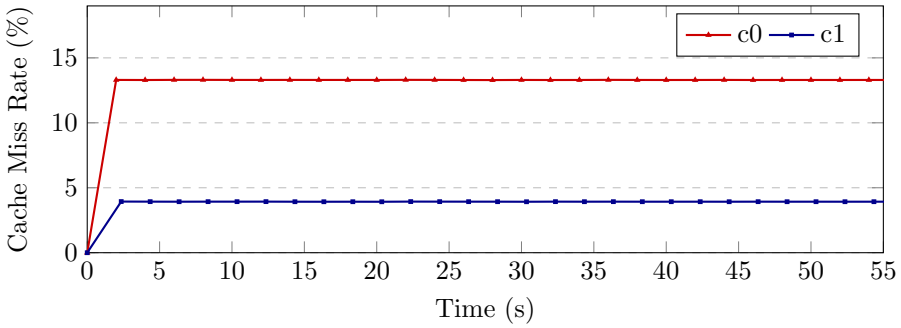


Figure 6.8: The containers were allocated with two shared and four exclusive cache ways each.

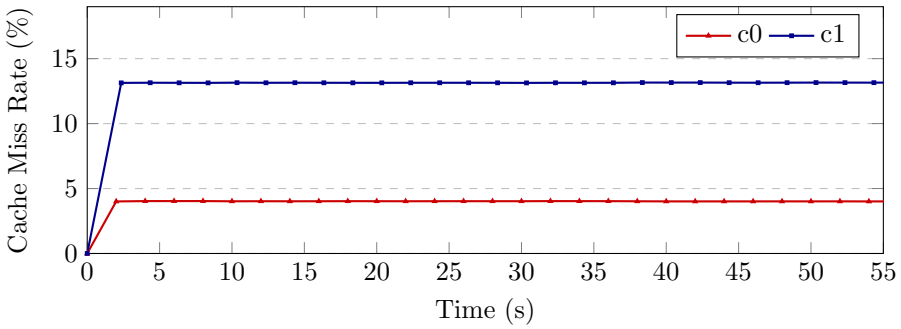


Figure 6.9: The cache miss rates of the reversed four-exclusive two-shared partition.

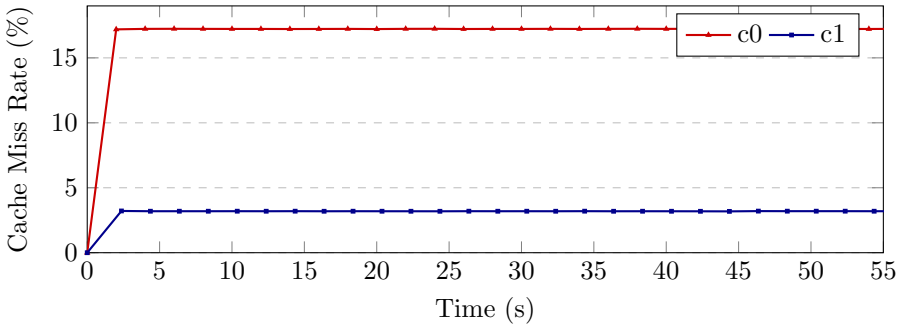


Figure 6.10: The per container when sharing four ways and using three in exclusive.

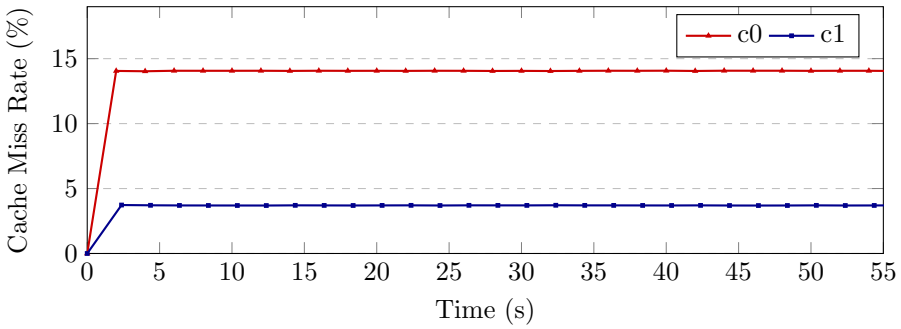


Figure 6.11: The cache miss rate for six shared cache ways and two in exclusive.

6.2 Probing in a Chaotic Environment with RDT

In this section, we will test the level of isolation that RDT cache partitioning can provide and how the online sensitivity probing can exploit it. Typically, meaningful online profiling is obfuscated by other workloads running on the same node. Since the behavior of these unrelated workloads is unpredictable, most approaches must bear the burden of offline profiling. The goal is to online-probe a single-core PostgreSQL database container using the probing implementation introduced in Section 4.5.2.

Figure 6.12 depicts the baseline result where the database is running alone and where we probe for sixty seconds. The database container is using 1 GB pages. We set up a noisy environment, which can be seen in Figure 6.13 and 6.14. In this example, we have 15 concurrently running containers that randomly start a data-intensive microbenchmark for a random time between 1 and 5 seconds, then idle for the same random time interval. If we run the experiment without isolation five times, we yield practically meaningless sensitivity results, as can be seen in Figure 6.15. The baseline measurement and the specific sensitivity measurements are most likely influenced by the other workloads.

The sensitivity order is correctly determined when we employ isolation via RDT, as shown in Figure 6.16. In this example, we isolate the database and the probing container in a 6-way and 50% memory bandwidth partition. This strict allocation slightly amplified the concrete values. While the former baseline probing returned an average last-level cache sensitivity of 8%, the reduced yet isolated probing yields an average of 11% for the same shared resource. This amplification continues with the 3-way 30% memory bandwidth configuration seen in Figure 6.17.

However, the order of sensitivities is reported correctly and consistently, which is a significant aspect for many optimization techniques, such as linear optimization. Nevertheless, the resource demand of applications is typically non-linear, as for example seen in the evaluation Section 7.3. Hence, it is important to choose an adequate and consistent isolation partition. Compared to probing in noisy environments, isolation enables new levels of online probing without

requiring a full socket or node.

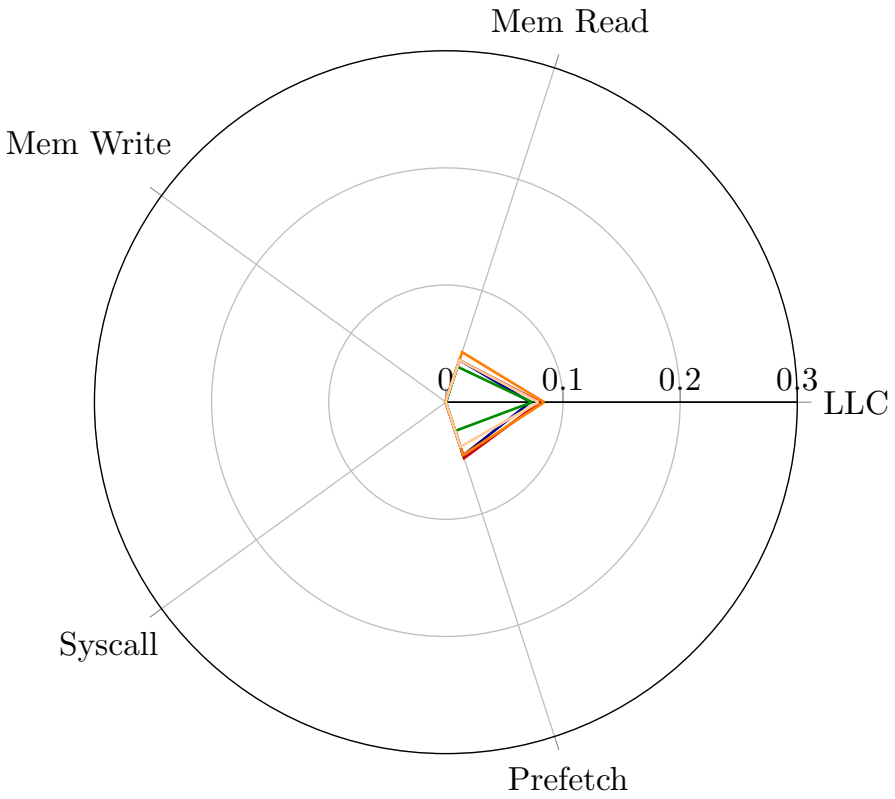


Figure 6.12: Five distinct sensitivity probing results of a PostgreSQL database running alone on a host. Especially the dimensions that we tightly control via RDT are reproduced accurately.

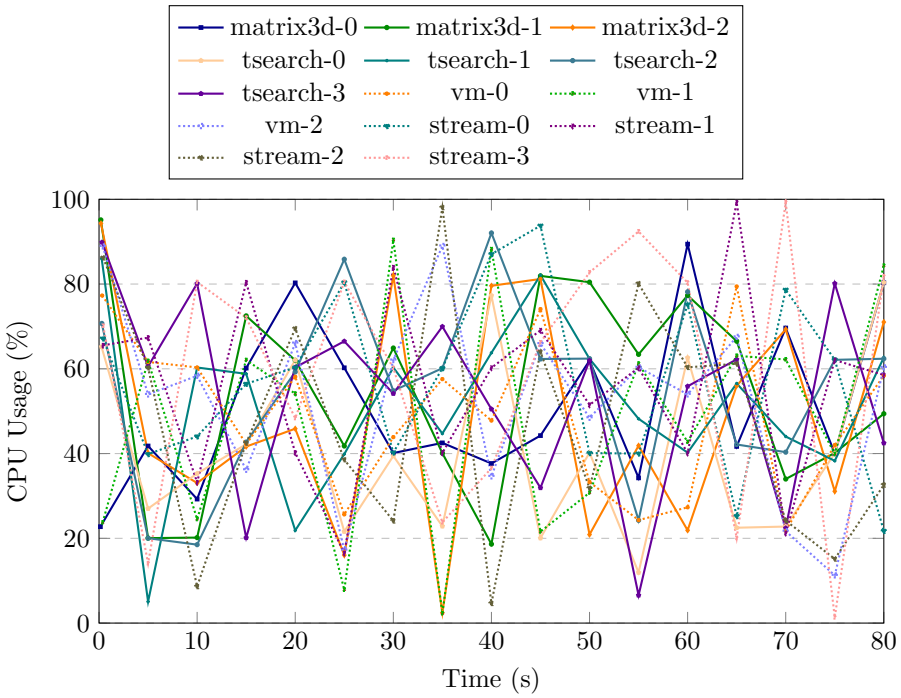


Figure 6.13: The CPU usage in a randomized probing environment.

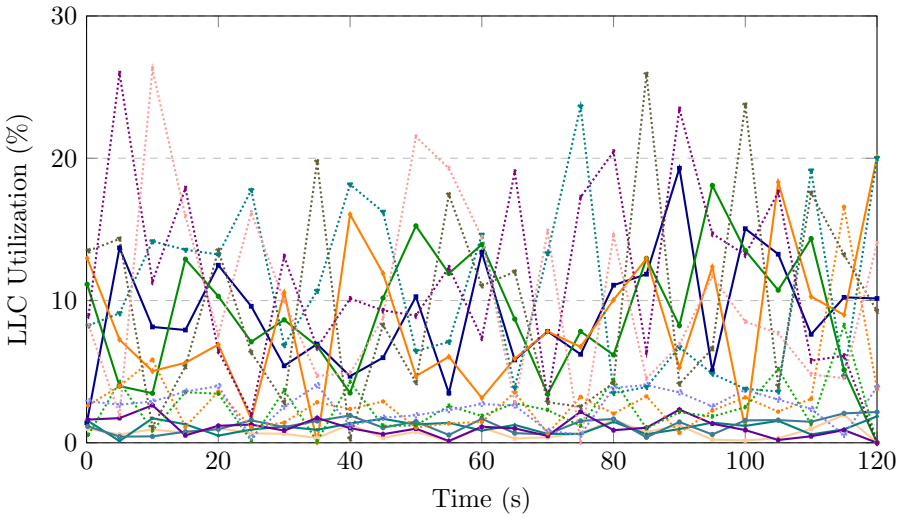


Figure 6.14: The last-level cache utilization in a randomized probing environment.

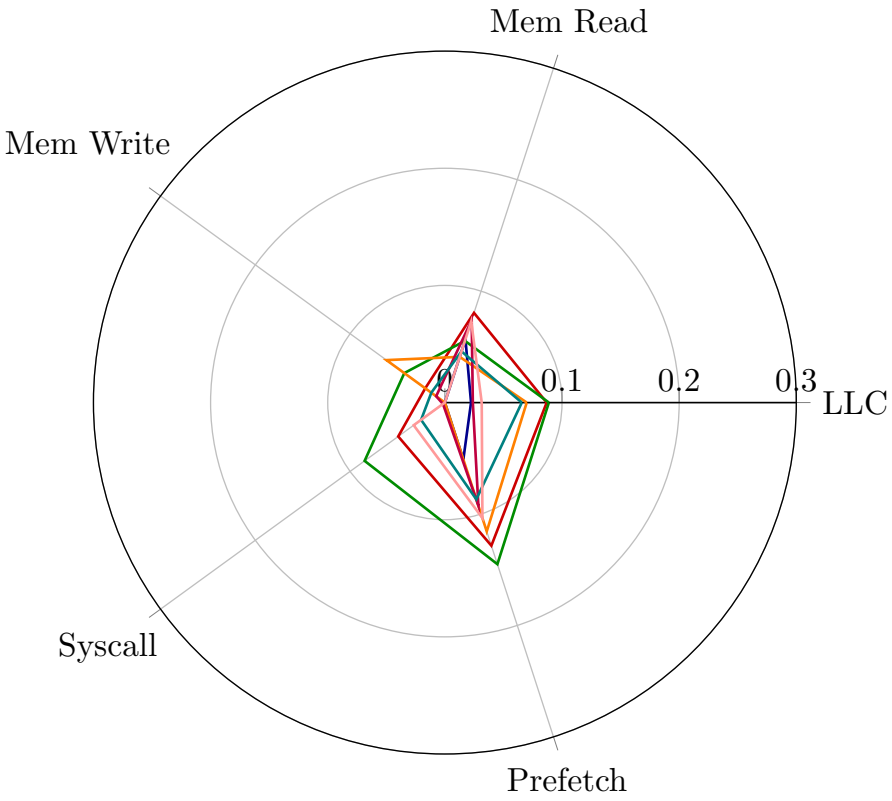


Figure 6.15: Multiple distinct sensitivity probing results of a PostgreSQL database which runs unisolated along random noise. The results cannot be interpreted in an optimization process as they are heavily influenced by the noise.

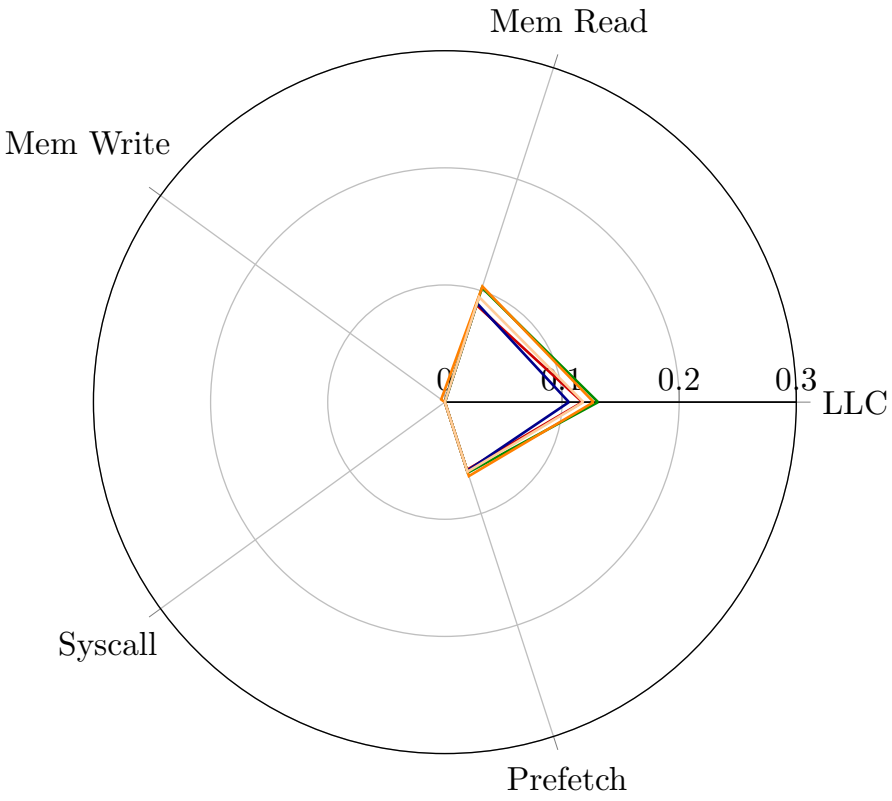


Figure 6.16: In this instance, we probe the database running with six cache ways and 50% of the available bandwidth. The isolation successfully restores the sensitivity order. Because of reduced resource availability, however, the concrete values are slightly higher.

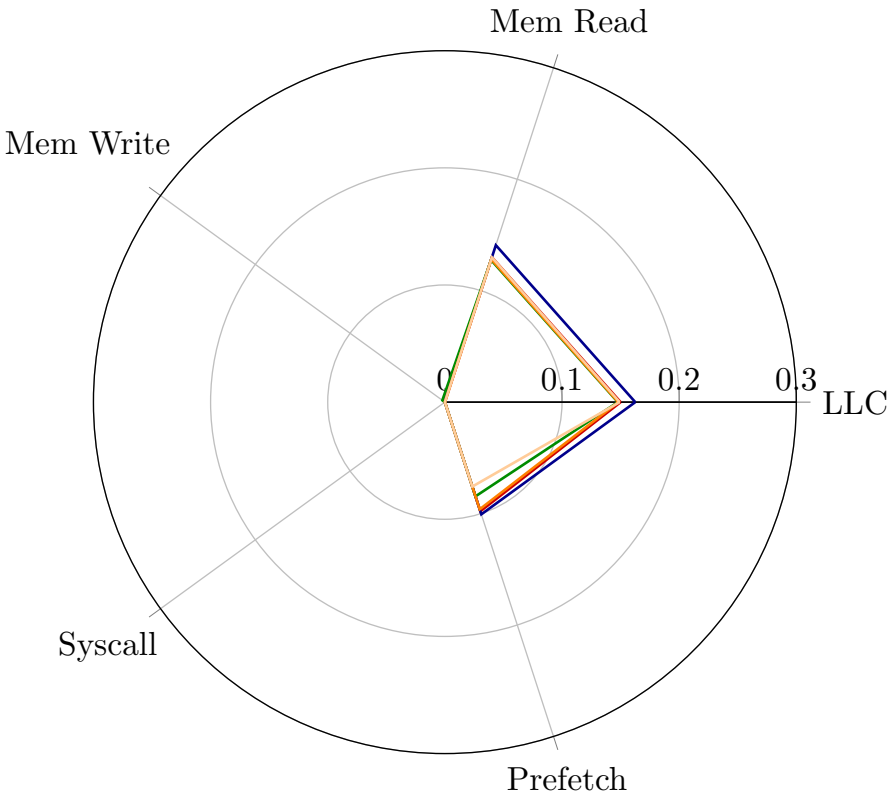


Figure 6.17: The PostgreSQL database and the respective probing container run in a resource allocation group which was granted three out of twelve cache ways and thirty percent of the available memory bandwidth. The configuration successfully restores the sensitivity order with amplified values.

Chapter 7

Evaluation

In this chapter, we evaluate the performance of the implemented scheduling policy. Further, we analyze the workloads described in Chapter 5 and discuss the patterns that emerge when utilizing RDT for interference-aware scheduling and probing. We ran all the related experiments on the formally described and validated Intel Xeon Silver 4314 testbed.

7.1 Scheduler

In this section, we will evaluate the performance of the dynamic scheduling policy implementation. The methodology of this policy was introduced in Section 3.4. As a baseline, we will compare it with the Least-Loaded policy, which is commonly used in task scheduling. We evaluate three load scenarios: High Utilization, Exceeding Demand, and Low Demand. As the percentage of priority jobs varies by scenario, so does the opportunity to allocate exclusive resources. The scenarios are detailed in Section 5.1.

Figure 7.1 shows the IPC Efficiency that was achieved by the schedulers in the different scenarios. This includes the global IPC Efficiency for all jobs and the specific number within the priority and batch job pools. In terms of global and priority IPCE, the dynamic scheduler achieves better results across all scenarios with minimal penalties to batch jobs, as shown in Figure 7.2.

In the High Utilization scenario, the efficiency of the priority jobs improved by 38.2% at a cost of 0.6% decreased batch job efficiency. Overall, this results in a 4.3% improvement globally. That amounts to a batch penalty to the global improvement ratio of 7.17, which is the best of all scenarios. This improvement is enabled by the nonlinear performance-to-resource relationships of priority jobs. An exclusive allocation to a batch job would only give it an incremental improvement. Consequently, removing its available resources only causes an incremental penalty. The priority flag, however, hints to the scheduler that this job will perform significantly better if allocated with a specific amount or combination of resources. Typically, this combination includes a cache partition

large enough to enable locality. In its current state, the implementation still needs the explicit hint to know that a given container is worth probing and allocating. However, this could also be derived in the future, e.g., from the container's real-time data frames.

The best overall improvement was yielded in the Exceeding Demand scenario with a global improvement of 4.9%. In this scenario, the demand for RDT resources from arriving priority jobs exceeds the available resources. In general, the IPCE of priority jobs is better. Therefore, the global IPCE values are generally higher when more priority jobs are configured for a scenario. However, since there are always priority jobs that can be probed and allocated by the dynamic scheduler, the utilization of resources for priority jobs drives the global efficiency while the penalty still remains small.

In the Low Demand scenario, only 10% of the jobs that arrive at the queue are priority jobs. Hence, there will be periods when demand for exclusive resource partitions is insufficient. Hence, more resources will remain in the general pool available to the batch jobs. While this lowers the respective penalty, there is less opportunity to unlock overall improvement. From a quality of service perspective, a low demand will allow basically all priority jobs to run close to their $IPCE^{target}$ for at least a part of their execution time.

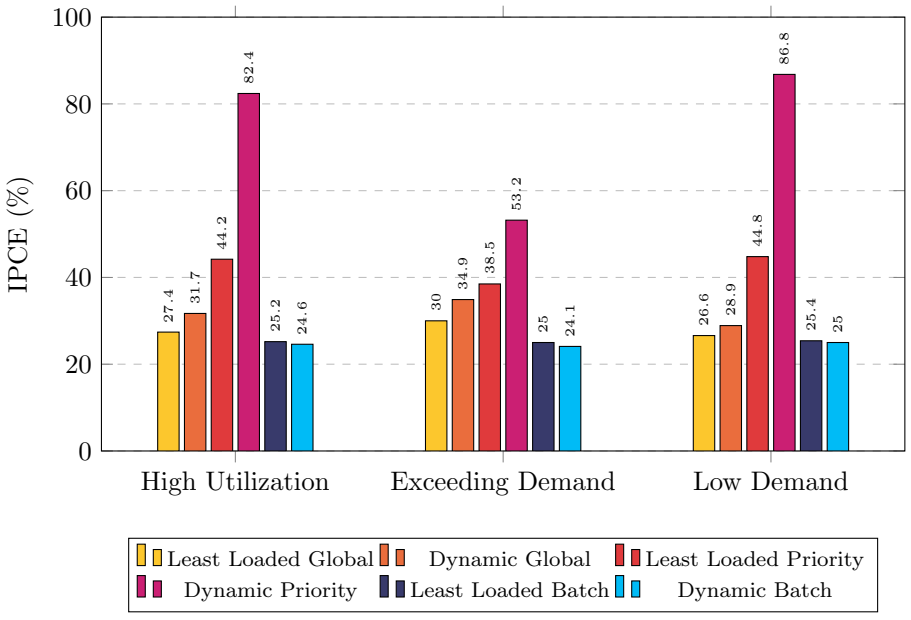


Figure 7.1: The IPC Efficiency values which were achieved globally, by priority jobs, and batch jobs.

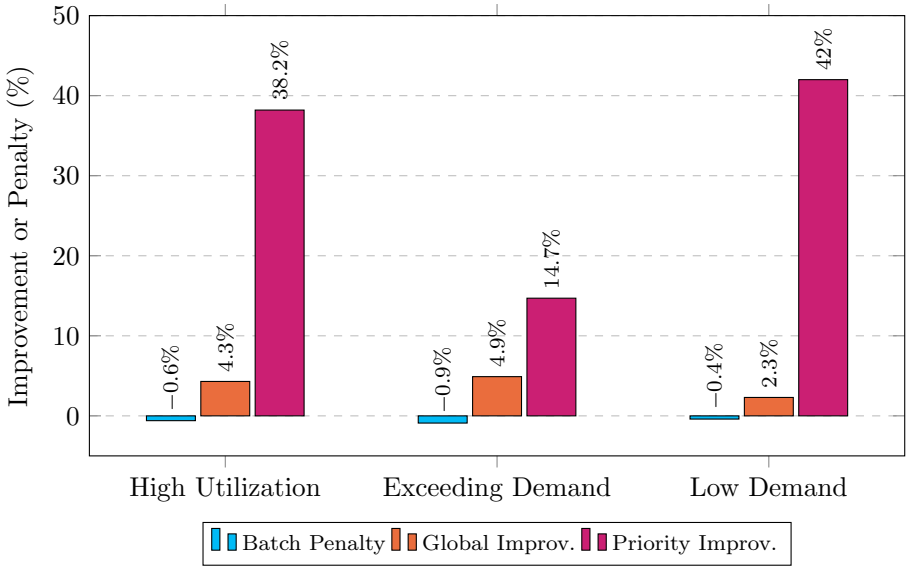


Figure 7.2: The improvements of the global and priority IPC Efficiency and the respective penalty to batch jobs.

The fraction of time during which priority containers can execute at or above their IPCE target depends on factors related to the load scenario and the scheduler's settings and capabilities. These include the percentage of priority jobs and the average number of RDT resources required per priority job. Further, the scheduler must wait a minimal warm-up time before probing a container. Likewise, there should be a cool-down between probings to allow the system to settle. In the evaluation benchmarks, we use ascending probing, which means the probing sweep starts with the minimum one-way/10% memory bandwidth allocation. At the beginning of the sweep, the IPCE will likely be low, which will hurt the overall values until it converges to the target.

Figure 7.3 shows the fraction and the 90th percentile for each scenario. As shown, even in the Low Demand scenario, the total fraction remains at 62.3% and the 90th percentile at 71.5%. While some of the time below the target IPCE can be attributed to the formerly described phases, intuitively, the values for the Low Demand scenario should be closer to 100%. A major factor contributing to the gap is brittle workloads. The probing of these workloads finishes just above their target with a small margin. Hence, they can periodically fall below their target, e.g., due to execution phases or subtle external interference that RDT cannot mitigate.

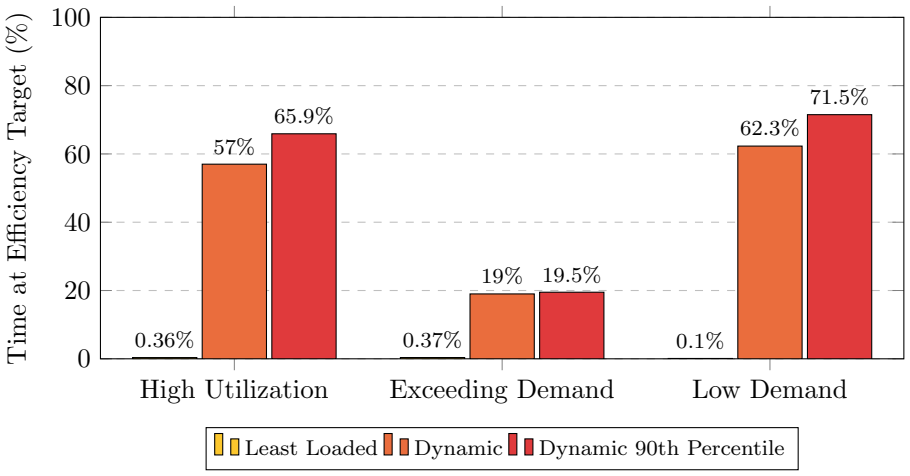


Figure 7.3: The time fraction where priority jobs met their efficiency target. For the Least Loaded policy, the only time when priority jobs can run efficiently is during the early admission period, e.g., if they arrive first.

Figure 7.4 shows a trace section of the Low Demand benchmark in which different priority workloads are started and executed. Each workload has an IPCE target of 90% indicated by the dashed horizontal line. As shown, most containers start with a poor IPCE of $\approx 20\%$. However, there are enough RDT resources available such that each container can be probed and allocated instantly after its warm-up period. Initially, the probe drives up the IPCE of the workloads by increasing their allocations. Once the target is found, the probe runner returns the allocation to the scheduler. The scheduler was configured to retain this allocation for the remainder of the job. However, some containers will again fall below the target, typically to $\approx 85\%$ IPCE. If we only consider the time above the target metric, e.g., if we pursue quality of service, this is obviously not beneficial. To this end, the scheduler can be configured with buffer ways and bandwidth, which adds a fixed additional amount of resources to the allocation returned by the probe runner.

While this buffering enables a higher quality of service, it significantly reduces the global efficiency by causing resources to be unused. If we allocate extra resources to improve stability, fewer containers can secure sufficient allocations to drive their IPCE above the tipping point. This means that fewer priority containers will execute significantly better, while the penalty to batch jobs remains roughly the same. Typically, the extra resources will mostly remain unused. This problem depends on the granularity of RDT partitions on the given host. In this testbed, each last-level cache way amounts to 2 MB. If a container needs 2.1 MB of cache memory to fully exploit locality, we have three options: if one cache way still yields a significant improvement, we can allocate it and run the container slightly below its optimum. This includes the

case where a container can have periods below and above the target. Second, we can overallocate it with two cache ways and accept that we leave 1.9 MB unused. Third, we can probe whether the workload can transfer the pressure from the cache to the memory bandwidth to compensate. Sharing certain cache ways can, in theory, mitigate the granularity problem to some extent. However, the concrete behavior when overlapping allocations can be less predictable, as demonstrated in the validation chapter.

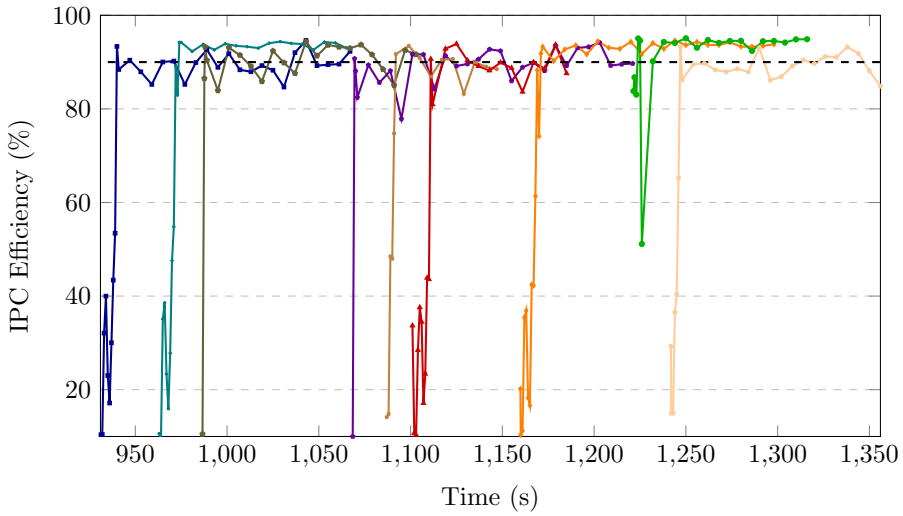


Figure 7.4: The IPC Efficiency of different priority containers during the Low Demand scenario. All containers are probed and allocated immediately, enabling 90% which corresponds to the target efficiency. However, the IPCE of some containers is brittle, meaning it can fall below 90% again.

Overall, the dynamic scheduler can reliably improve efficiency and, hence, throughput. Generally, there is more optimization opportunity if sufficient priority jobs are available. As a benefit, the allocation probing is fully online and specific to the host. This means the policy works independently of the host and needs no further configuration. Former RDT integrations, such as the Kubernetes Controller [56], are static and require manual configuration per host.

However, the policy implementation is subject to two main limitations of RDT. First, workloads that do not support 1 GB pages require significant extra resources. Second, the implementation needs to carefully handle the inner and outer fragmentation of RDT resource partitions. Inner fragmentation refers to unused resources within a partition. This occurs when the smallest allocatable partition does not evenly divide the workload's demand and hence must be overallocated. Additionally, inner fragmentation can be temporal if a workload

only uses the resources periodically. In this case, the scheduler would need to detect and handle these periods. The outer fragmentation concerns only the cache-way partitions, specifically the continuity constraint on allocated cache ways. This requires frequent consolidation when workloads finish and relinquish their resources. Repacking the partitions to enable new, contiguous allocations can impact ongoing workloads as the data in their former cache ways will likely be evicted.

7.2 Allocation Probing

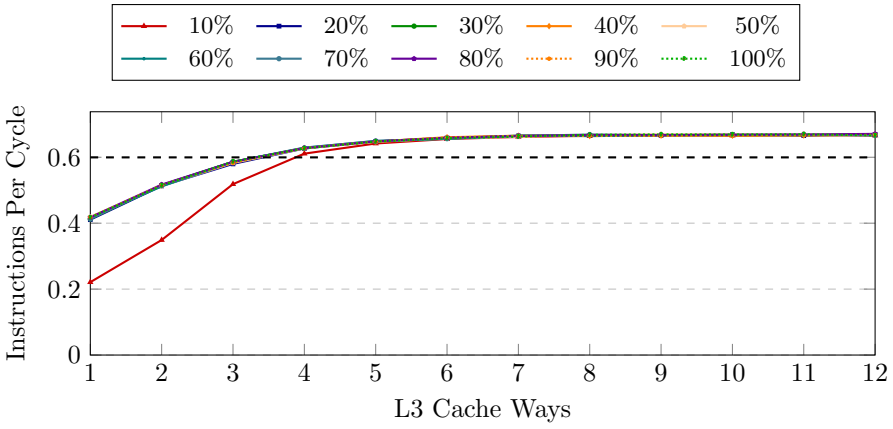
In this section, we present how the developed probing process can identify the non-linear resource demand of different applications using RDT. Furthermore, this approach facilitates the identification of an adequate or optimal allocation in scheduler implementations on any given host with RDT. We use the `AllocationProbe` implementation to perform a full sweep of the possible allocation combinations.

To this end, we display the Instructions Per Cycle (IPC), the IPC Efficiency, and the overall cache miss per allocation combination of 1–12 cache ways and 1–100% memory bandwidth. We enumerate benchmark examples that represent general patterns that we identified. The dashed horizontal line represents 90% of the best value found during the allocation sweep.

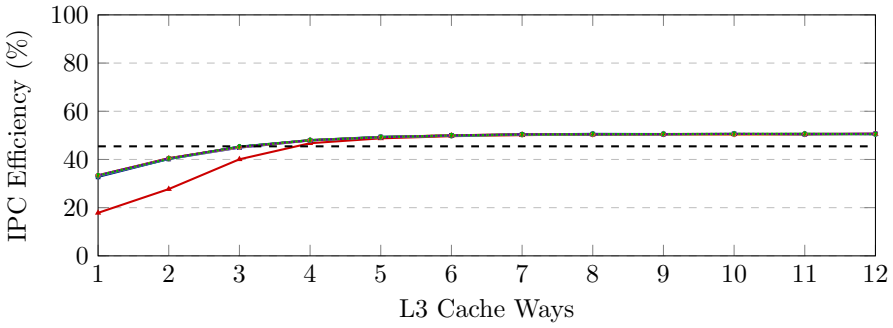
7.2.1 Locality and Streaming

The stress-ng microbenchmark `tsearch` showcases how an application that is able to establish locality can have a non-linear improvement once its resource partition is large enough, as can be seen in Figure 7.5. The benchmark implements a tree search on a data set of 2^{17} nodes. Once the container is allocated with at least eight cache ways, totaling 16 MB of exclusive cache on this testbed, it incurs virtually no cache misses, thereby driving IPC and IPCE. Regarding the memory bandwidth, once the tree resides in the cache, the gap between 10% and the others is closed. In Figure 7.6, we run the same benchmark with 2^{20} nodes. In this case, the container cannot establish locality. It can transfer some of the pressure created from the lack of cache capacity to the memory bandwidth. This is indicated by the significantly worse performance across all 10% memory-bandwidth combinations.

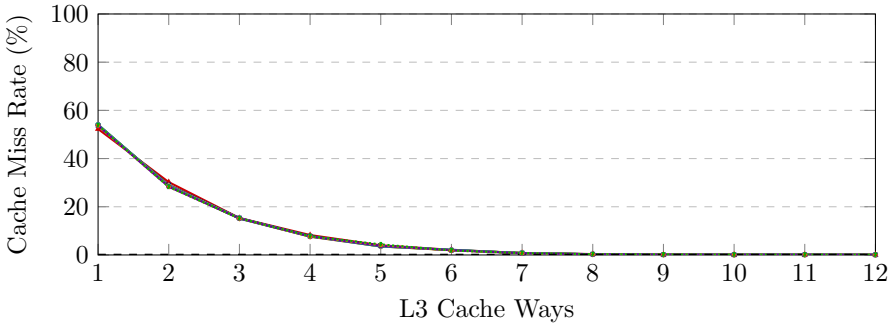
The 10% memory bandwidth allocation is inconsistent and struggles to fill the partition. Beyond the 20% bandwidth allocations, no further improvement is observed. This is likely because the next load operation of the tree search depends on the current one. A larger bandwidth does, for example, not provide more prefetching opportunities. The `tsearch` microbenchmark represents workloads that are strictly dependent on a cache partition that is at least as large as its working data set.



(a) IPC

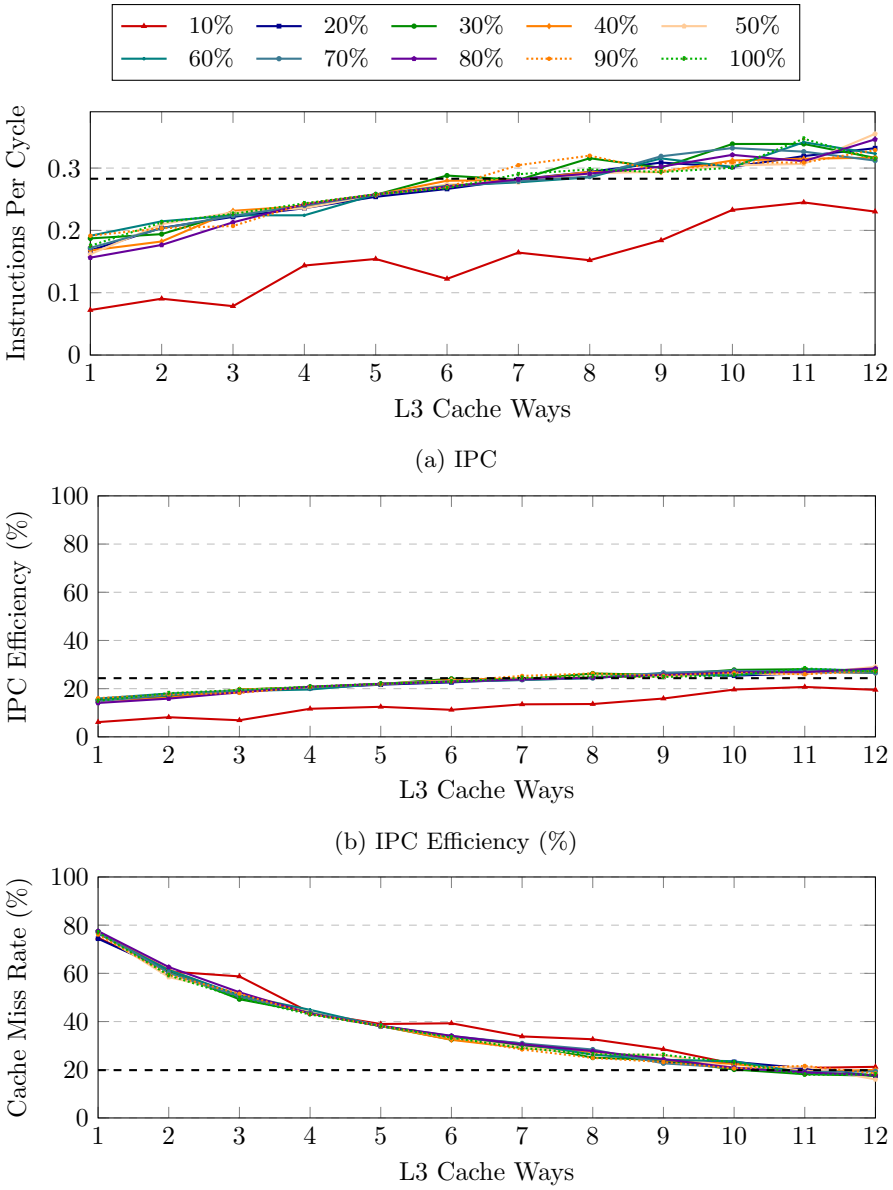


(b) IPC Efficiency (%)



(c) Cache Miss Rate (%)

Figure 7.5: Metrics for different allocation combinations for tree search with with 2^{17} nodes.



(a) IPC

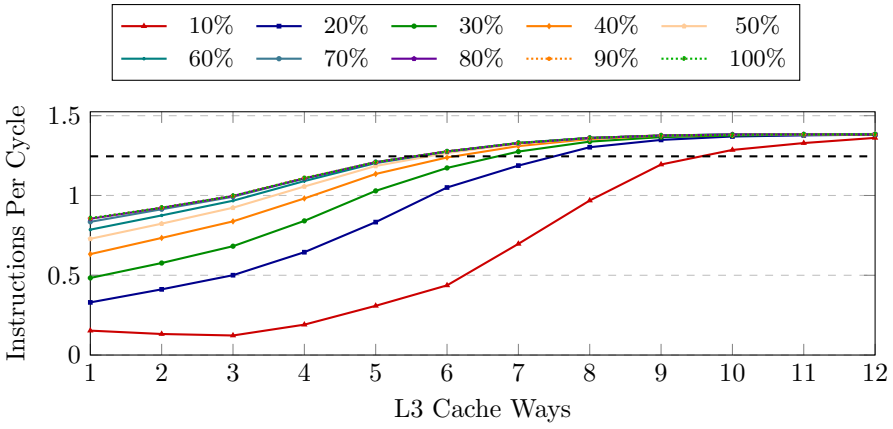
(b) IPC Efficiency (%)

(c) Cache Miss Rate (%)

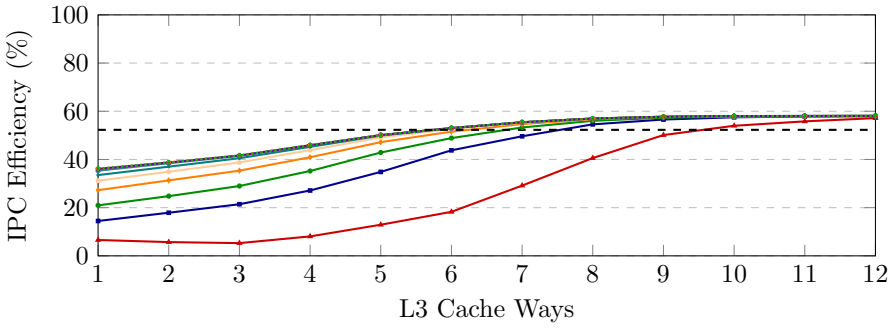
Figure 7.6: Metrics for different allocation combinations for tree search with with 2^{20} nodes.

Figure 7.7 shows the `matrix3d` benchmark with an input size of $96 \times 96 \times 96$. With the exception of three cache ways, increasing the number of allocated cache ways reduces the number of cache misses the container incurs, thereby improving performance. The ≈ 10`MB` of the three active $96 \times 96 \times 96$ 32-bit float matrices could mostly fit into the cache partition starting with five exclusive ways. However, we must consider the effects of partitioning on conflict misses, as discussed in Section 6.1.

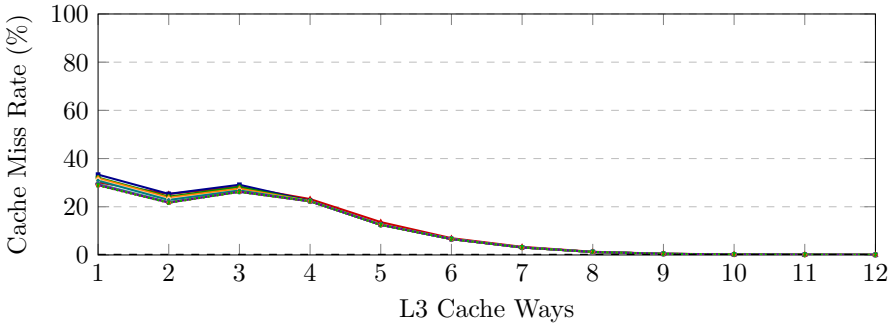
Until it can establish full locality and even slightly beyond, its performance is strongly impacted by the bandwidth allocation. This effect is amplified for the $128 \times 128 \times 128$ variant seen in Figure 7.8. The data must be constantly loaded into the cache and written back to memory upon eviction. However, the `matrix3d` benchmark exploits spatial locality because the rows of the matrices are partially processed sequentially. The `matrix3d` benchmark, therefore, represents a class of workloads that stream large amounts of data and hence will likely not fit all of it into any cache partition. As a consequence, they are heavily dependent on memory bandwidth, particularly when data is consistently prefetchable.



(a) IPC

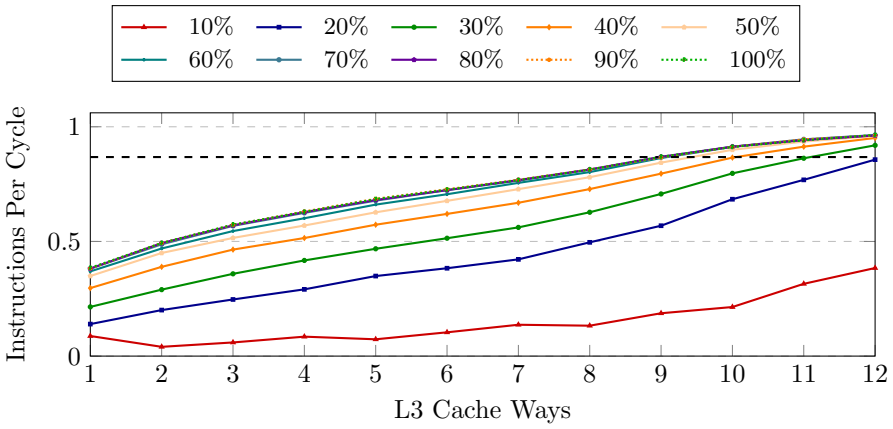


(b) IPC Efficiency (%)

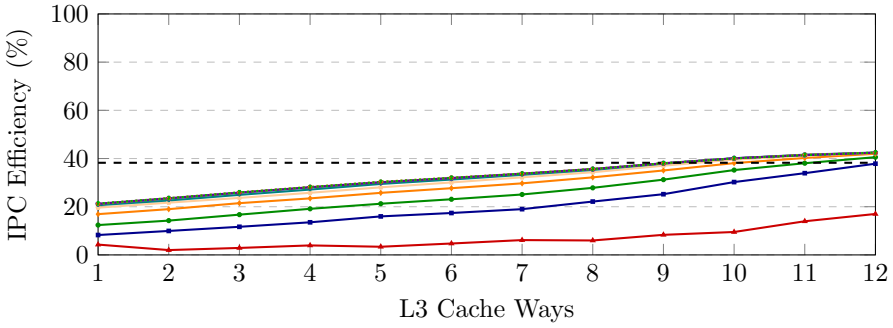


(c) Cache Miss Rate (%)

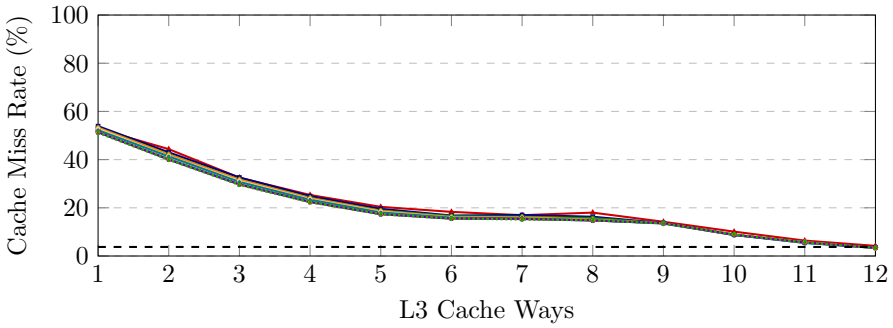
Figure 7.7: Metrics for different allocation combinations for processing a 3D Matrix of $96 \times 96 \times 96$ entries.



(a) IPC



(b) IPC Efficiency (%)



(c) Cache Miss Rate (%)

Figure 7.8: Metrics for different allocation combinations for processing a 3D Matrix of $128 \times 128 \times 128$ entries, the default size of the `stress-ng` 3D Matrix benchmark.

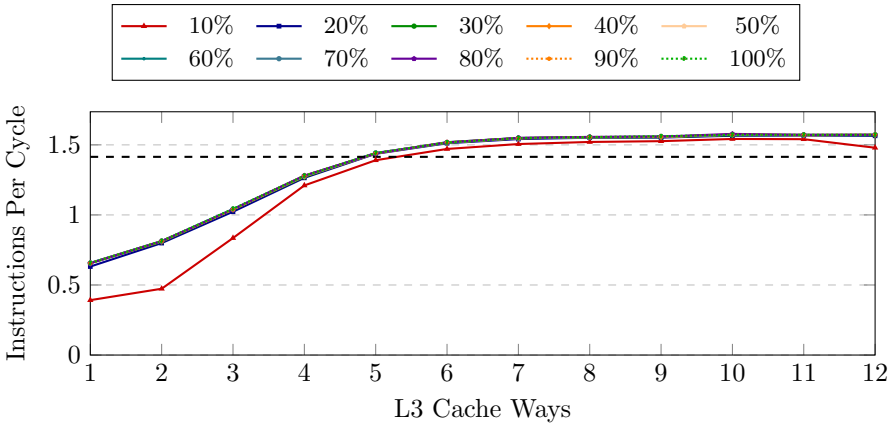
7.2.2 Configuration Impact & Phasing

The compression benchmarks are excellent examples of how the same application can shift its resource demand depending on the chosen configuration. Furthermore, it shows that relying solely on IPC as a metric can yield inconsistent results across applications that use different instructions depending on their execution phases. Figure 7.9 shows the allocation combinations of 7-Zip's compression level two setting. As in the previous tree search example, its performance depends solely on the size of its last-level cache partition.

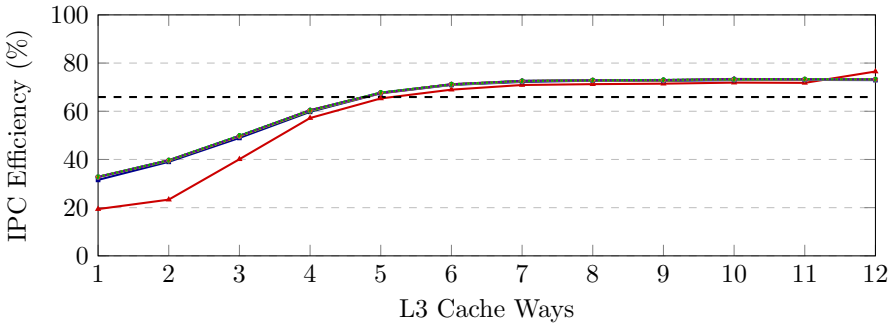
For compression level five depicted in Figure 7.10, any allocation with more than 10% memory bandwidth achieves the maximum possible performance on this system. A key difference between the two levels is likely the size of the dictionary [57] which is used. For levels below 3, the dictionary is typically less than 1 MB. Level five uses a much larger dictionary ($\approx$ 16 MB), which will likely never fit within any partition, especially since the data to be compressed must also be loaded. Figure 7.11 shows the highest compression level, nine.

Its IPC values are contradictory: more cache yields lower performance. Depending on the 7-Zip version, the dictionary size is approximately 64 MB, which limits locality. Additionally, the highest level performs more passes over the files to be compressed. Depending on the execution phase, the compression algorithm also uses more or fewer system calls to read and write to files.

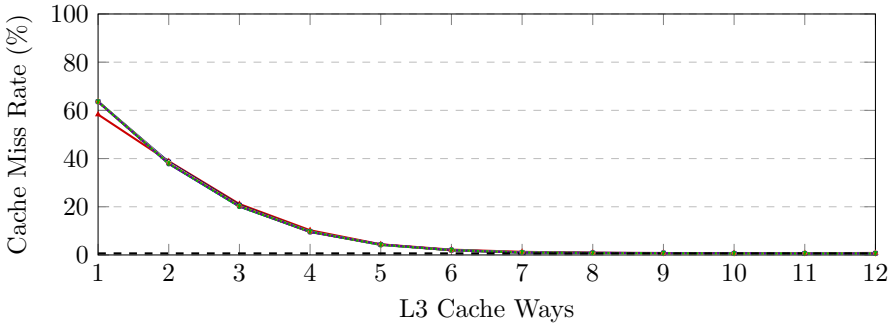
As shown in Figure 7.12, the performance bottleneck shifts from memory to branch prediction. Over time, the branch misprediction rate converges to $\approx 3\%$ which has a proportional impact on IPC. This increase in branch misprediction is subject to the implementation and unrelated to allocated RDT resources. Since the lower allocation combinations are tested earlier, the probing indicates that fewer resources are better. This effect is amplified by the instructions executed during each phase of execution. Multi-cycle instructions that do useful work contribute less to the IPC. The IPC efficiency metric considers the stalls, which are safer in this case. However, stalls caused by branch misprediction or, more generally, by events outside our control should be excluded from the IPCE to mitigate the remaining error in future work.



(a) IPC

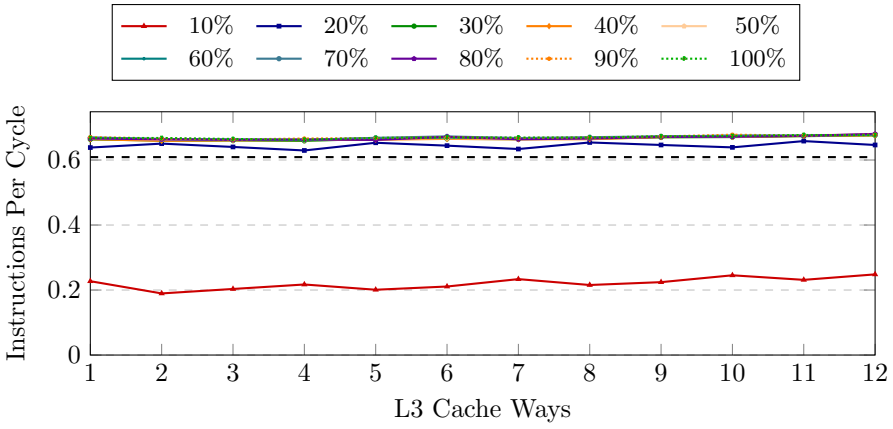


(b) IPC Efficiency (%)

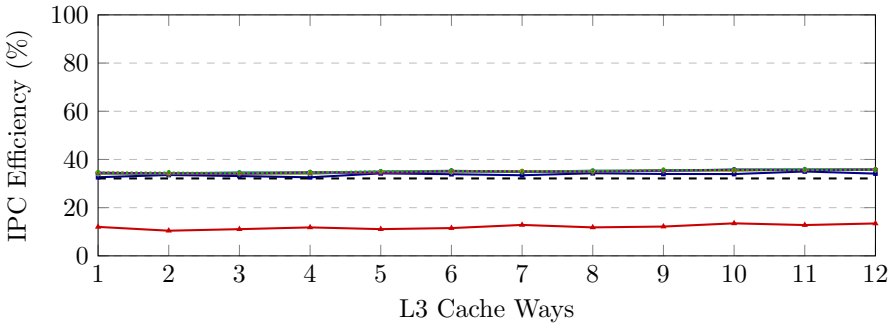


(c) Cache Miss Rate (%)

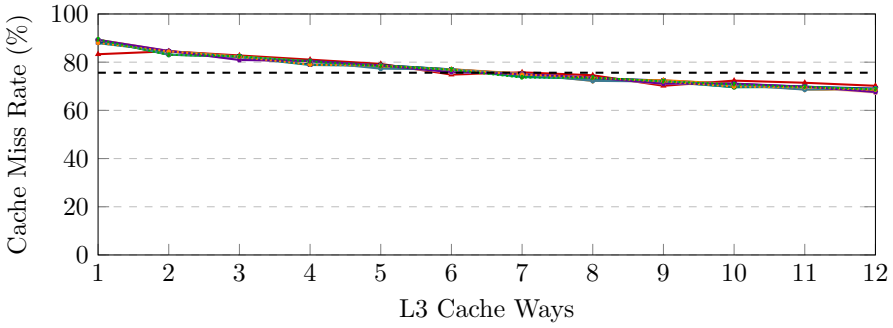
Figure 7.9: Metrics for different allocation combinations for single core compression with compression setting two.



(a) IPC

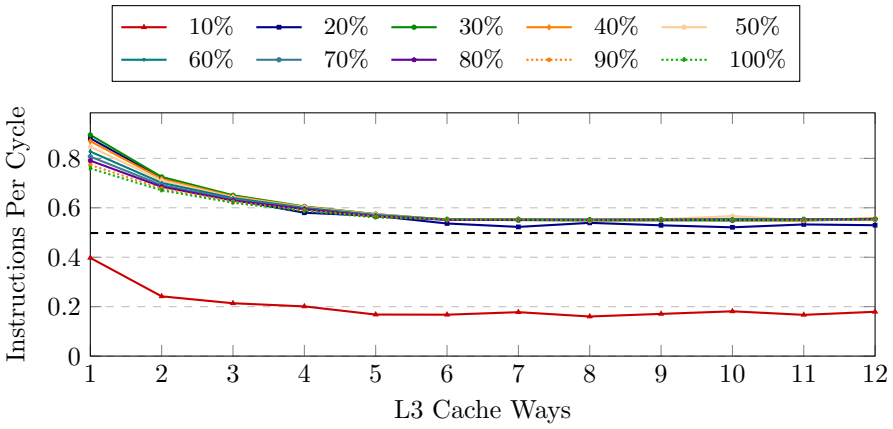


(b) IPC Efficiency (%)

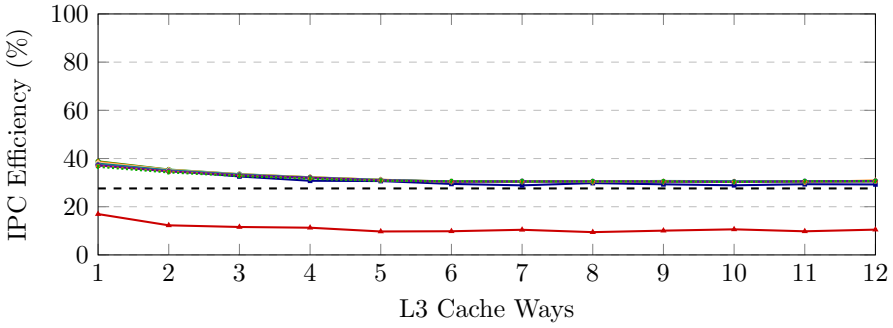


(c) Cache Miss Rate (%)

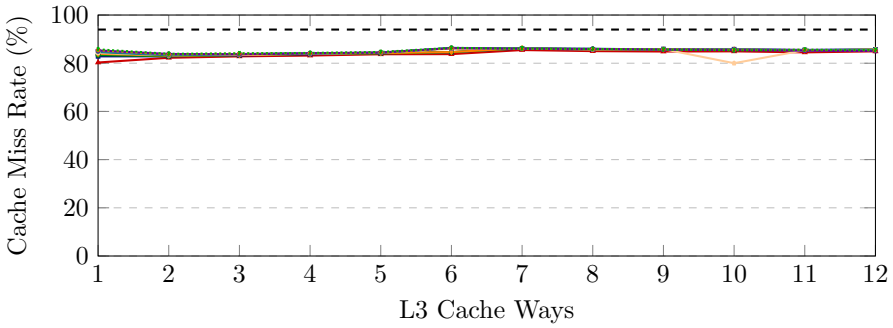
Figure 7.10: Metrics for different allocation combinations for single core compression with medium compression level five.



(a) IPC



(b) IPC Efficiency (%)



(c) Cache Miss Rate (%)

Figure 7.11: Metrics for different allocation combinations for single core compression with the highest level nine. The IPC metric fails to accurately reflect the impact of allocations due to the phasing.

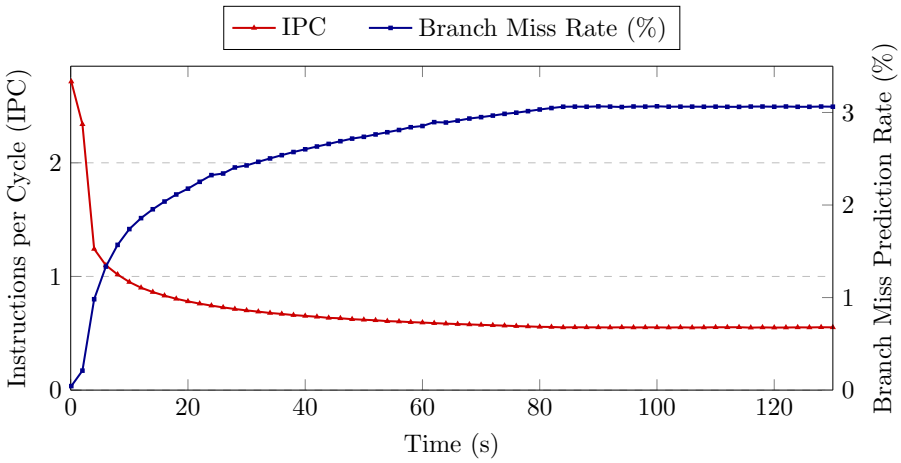
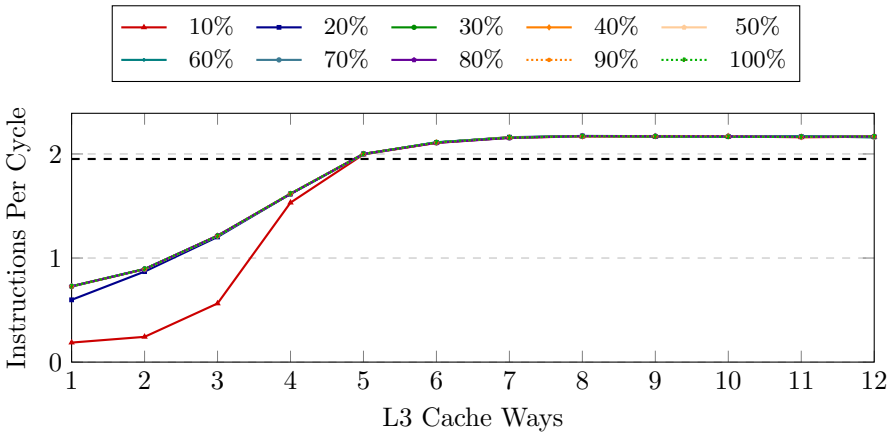


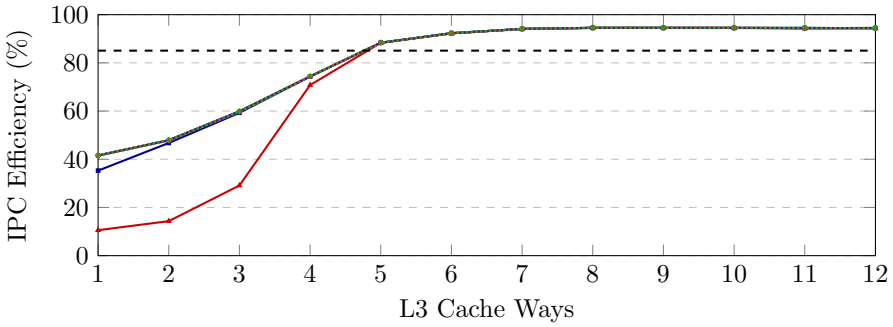
Figure 7.12: The Instructions per Cycle (IPC) of a single core compression job with the highest setting nine.

7.2.3 Impact of Page Sizes

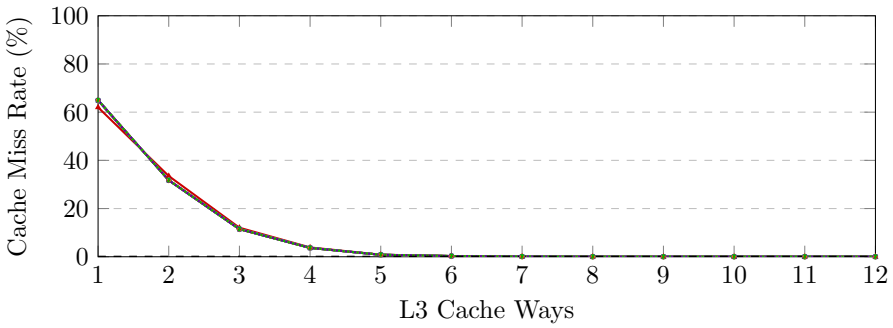
The allocation curve for the neighbor benchmark once again demonstrates the impact of page sizes on RDT partition allocation. In both Figures 7.13 and 7.14, the container is randomly looking up a buffer of 6 MB. The container that uses 2 MB pages requires a 6–7 way partition to avoid cache misses. This corresponds to a partition of 12–14 MB. Unlike this, the container backed by 1 GB pages establishes locality as soon as it runs with a 3–4 way partitions, which is equivalent to the size of its working set. When considering RDT for interference-aware scheduling and partitioning, it is therefore much more practical to focus on applications that support 1 GB pages, as they increase the utilization within a partition.



(a) IPC

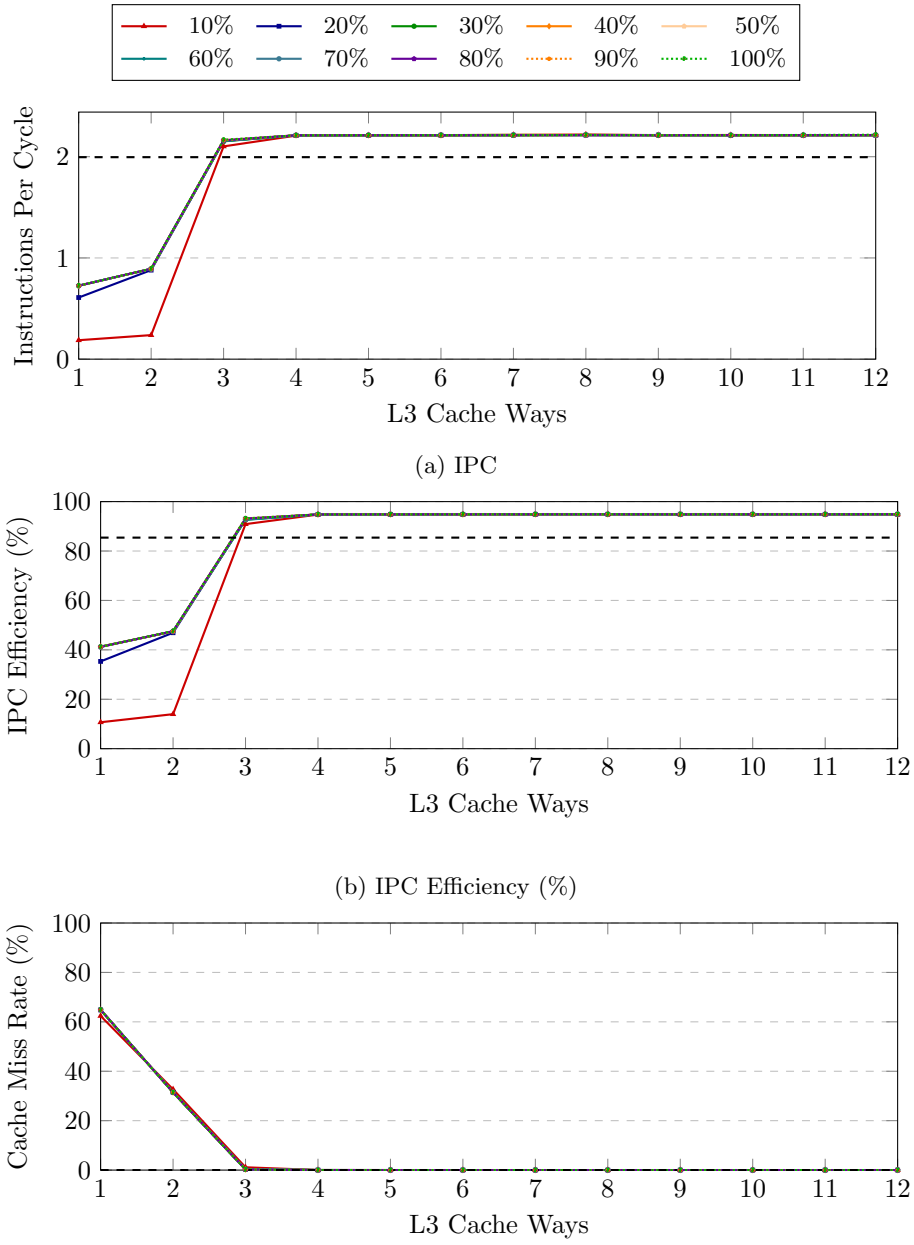


(b) IPC Efficiency (%)



(c) Cache Miss Rate (%)

Figure 7.13: Metrics for different allocation combinations for the `neighbor` applications using 2 MB pages working on a 6 MB buffer in random mode.



(a) IPC

(b) IPC Efficiency (%)

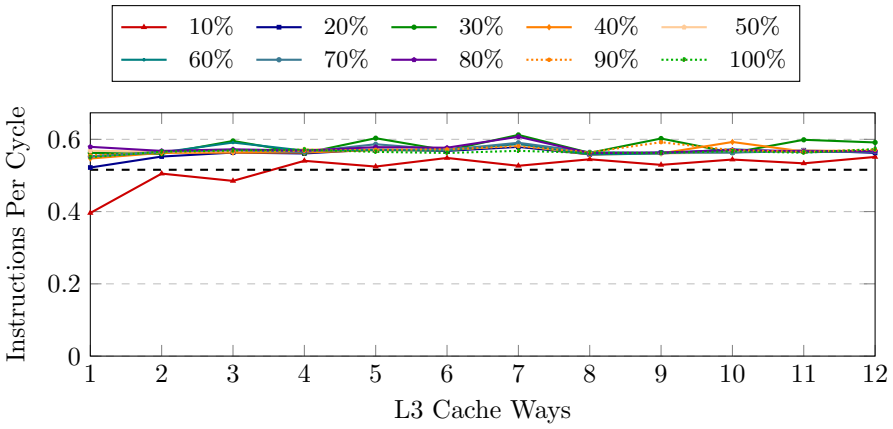
(c) Cache Miss Rate (%)

Figure 7.14: Metrics for different allocation combinations for the `neighbor` applications using 1 GB pages working on a 6 MB buffer in random mode.

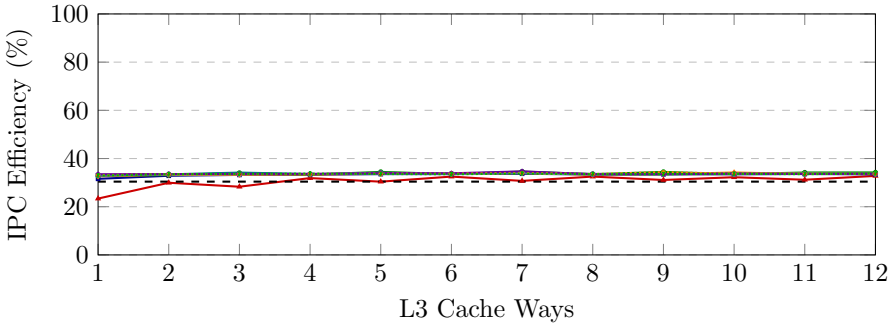
7.2.4 Higher-Level Resources

Figure 7.15 shows the results of different allocation combinations for a 4-core PostgreSQL container that handles mixed queries to a large dataset. The load for this benchmark is generated on a separate host. Hence, the results are also subject to the general performance of higher-level resources, such as the host's network interface card. Furthermore, the database can enter different spin lock phases, e.g., while waiting for transaction acknowledgments. This can influence the IPC metrics to a greater extent, for example, the IPCE metric.

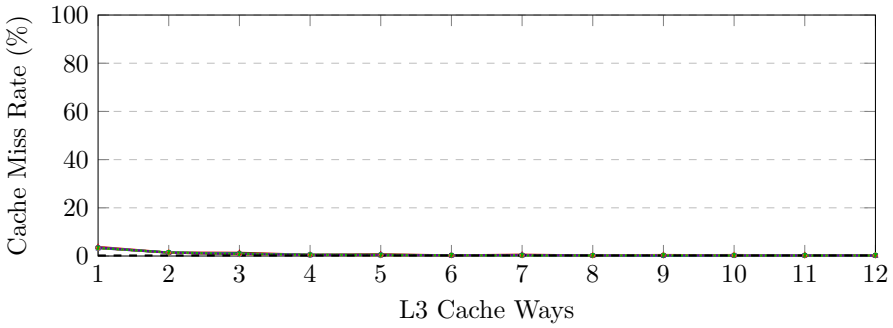
The PostgreSQL benchmark represents a class of networked workloads that *can* have higher-level bottlenecks. The impact of these resources typically depends on the network topology, the load generator's location, and the host's relative position. If the load generator is located on the secondary socket on the same node, the RDT resources determine the performance. If the load generator is on a different host and network, the intervening networking hardware can quickly become the bottleneck. In practice, this adds the dimension of load origin to the decision-making of an interference-aware scheduler.



(a) IPC



(b) IPC Efficiency (%)



(c) Cache Miss Rate (%)

Figure 7.15: Metrics for different allocation combinations for PostgreSQL running on four cores and a large dataset. The database is queried with mixed operations.

7.3 Sensitivity Probing

In this section, we will discuss the results of the sensitivity probing. The process itself was not used in the optimization policies because it is more useful in a multi-node scenario, where a scheduler has more opportunities to balance multidimensional interference sources. However, the results show that RDT enables reproducible, meaningful probing by isolating the target container from the rest of the system noise. Therefore, it would no longer be necessary to hold an exclusive node for probing, as is done in other research approaches. Additionally, the process works fully online.

In the following examples, we present the distinct patterns related to the sensitivity probing. To this end, we compare the sensitivity metric defined in Section 3.2.1. Overall, the probing reproduces accurately. However, the intersection of memory read and prefetching is quite high, as they imply each other. Similarly, the stressor targeting write bandwidth and kernel contention needs to be improved. In these examples, we use a single core to launch the stressors. The effects can be amplified at the cost of using multiple cores, e.g., by increasing memory write contention.

7.3.1 Microbenchmarks

Figure 7.16 depicts the sensitivity results of the `matrix3d` microbenchmark for different matrix sizes. The results confirm the conclusion of the previously described allocation probing: the sensitivity heavily depends on whether the workload can still exploit locality under pressure. This is the case for $16 \times 16 \times 16$ configuration where the working set is small enough to reside and be prefetched to cache, even while the stressor is putting contention on the respective resources. The configurations that are most affected by contention, however, are those that formerly exploited locality, e.g., the $96 \times 96 \times 96$ example. In contrast, the absolute penalty of containers that already incurred cache misses while they had the resources exclusively is relatively small, e.g., the $256 \times 256 \times 256$.

Similarly, the `tsearch` microbenchmark seen in Figure 7.17 follows this pattern. However, the penalties are much higher. Since each step in the tree search depends on the result of the previous one, the microbenchmarks suffer from increased latency, which occurs when the working set can no longer reside in cache memory under contention. This is the case for working sets starting from 2^{15} nodes, which corresponds to ≈ 1.6 MB of data, including overheads.

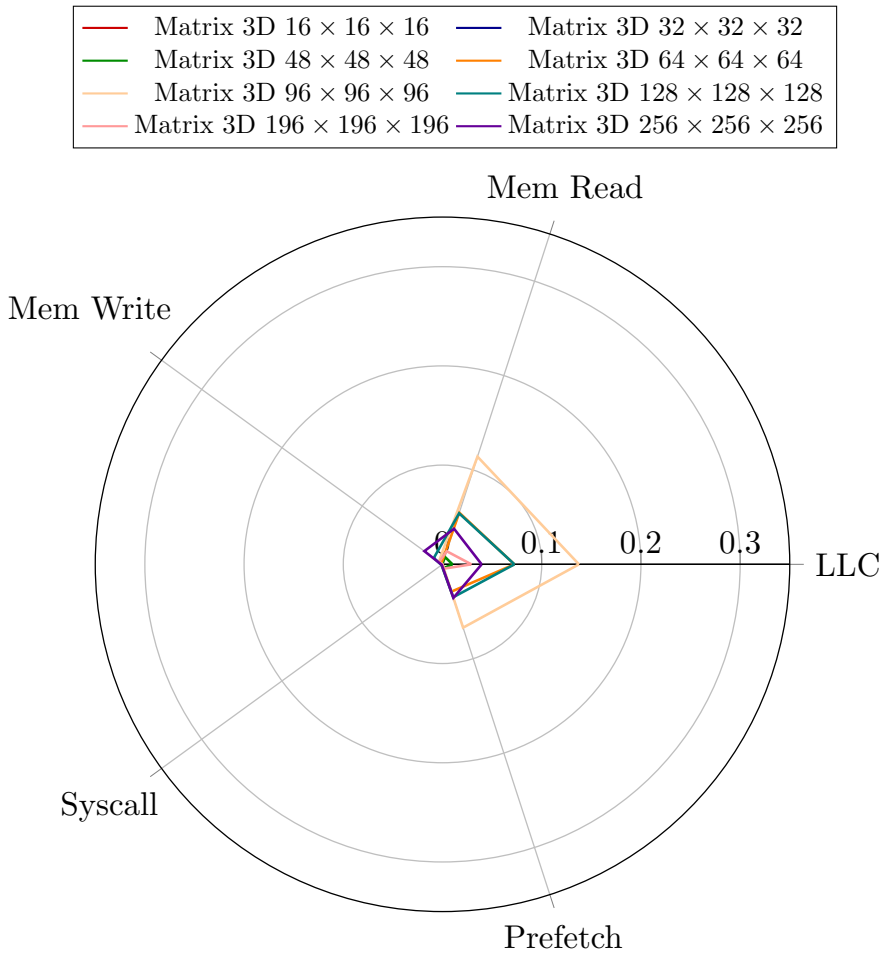


Figure 7.16: The IPCE of processing different 3D Matrices using the `matrix-3d` microbenchmark.

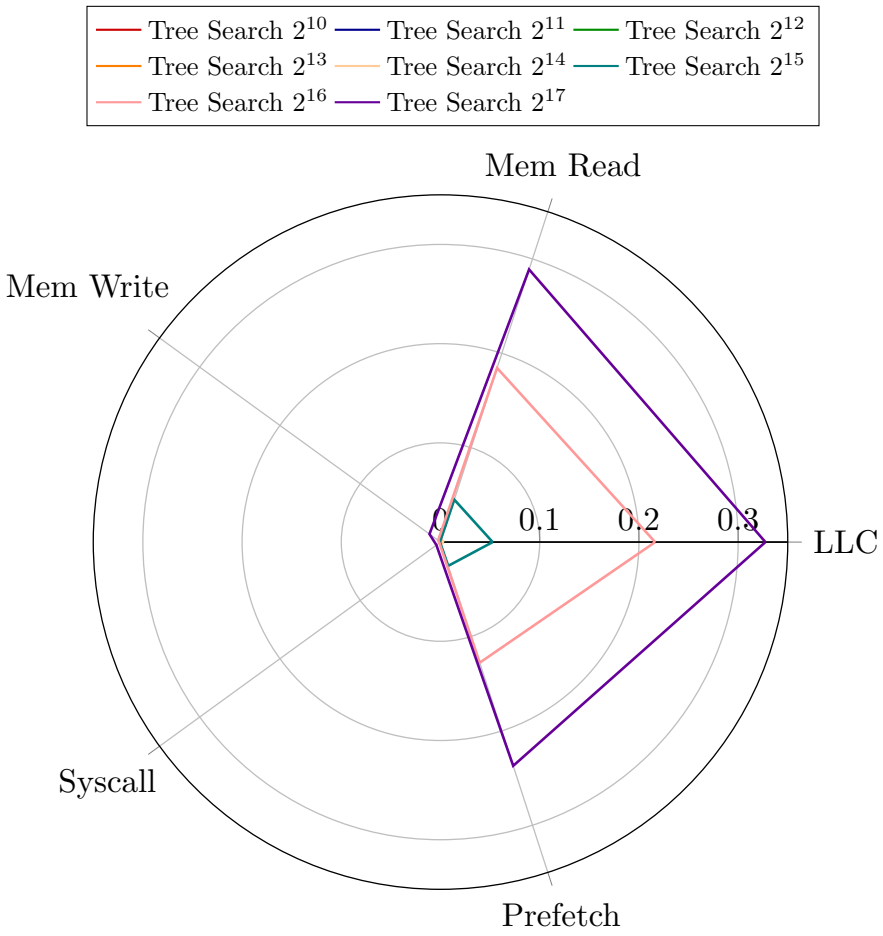


Figure 7.17: IPC Efficiency of binary tree search implemented in stress-ng `tsearch` microbenchmark.

7.3.2 Higher-Level Resources

Figure 7.18 shows the measured sensitivities of different PostgreSQL benchmarks. As can be seen, the results lack entropy. To minimize interference with the silicon's shared resources, the benchmarking load was generated on a different host in the same data center. Hence, the requests and transaction acknowledgments are transferred over the local network. The latencies are therefore mostly determined by higher-level resources, such as network cards or network switches. Hence, the sensitivity-probing degenerates into database probing while waiting for slow I/O. This causes very similar results. However, the outlier mixed operations in the small-dataset benchmark trial are consistently reproducible.

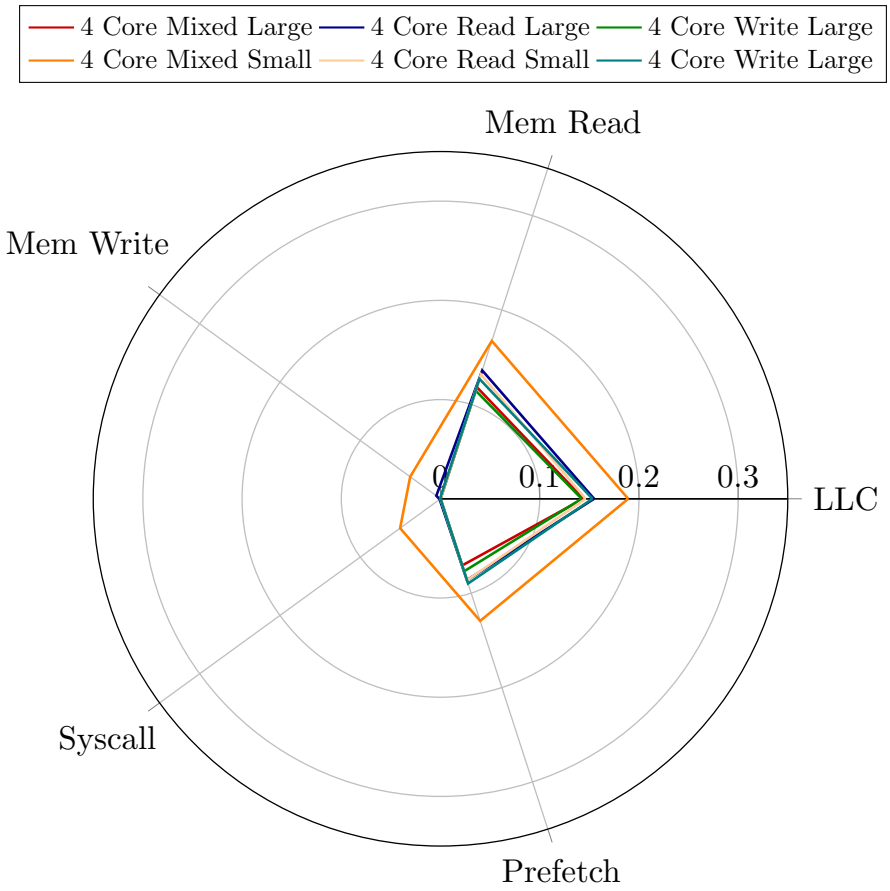


Figure 7.18: IPC Efficiency of PostgreSQL using four cores for a large dataset.

7.3.3 Compression

Figure 7.19 shows the sensitivities of different 7-Zip compression settings. Each compression level can affect a different compression algorithm used during the workload. Similarly, the size of the dictionary allocated during execution heavily depends on the chosen compression level and, in turn, impacts the container's cache memory demand. Without contention in the memory system, the lower-level ones and two can still fit or continuously prefetch their dictionaries into cache. Beyond these levels, the penalty will be less on this given host.

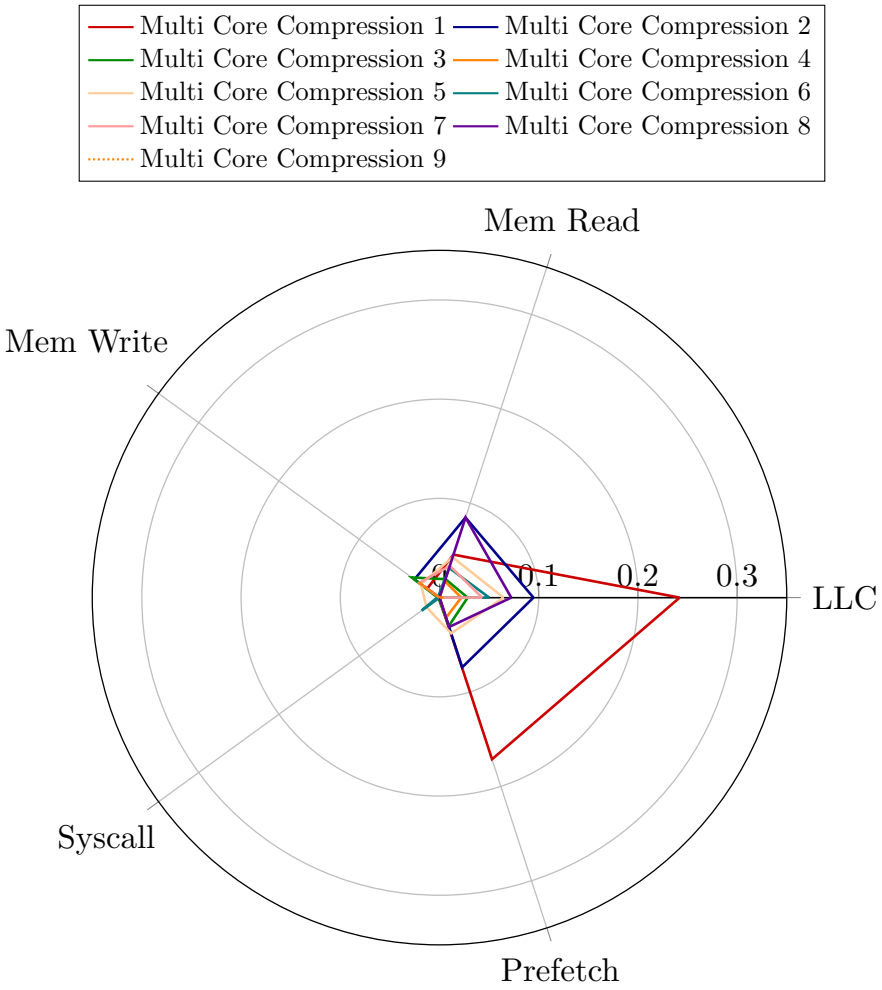


Figure 7.19: IPC Efficiency of different compression levels running on four cores.

7.3.4 Compilation

The GCC compilation result shown in Figure 7.20 serves as an example for collaborative multi-core jobs. In the 8- and 12-core configurations, the individual processes run at lower overall performance than in the single- and 4-core configurations. Hence, the impact of the contention on them is less as they already had to share the resources while the baseline was established. However, the processes will likely benefit from shared instructions and from data being loaded into cache.

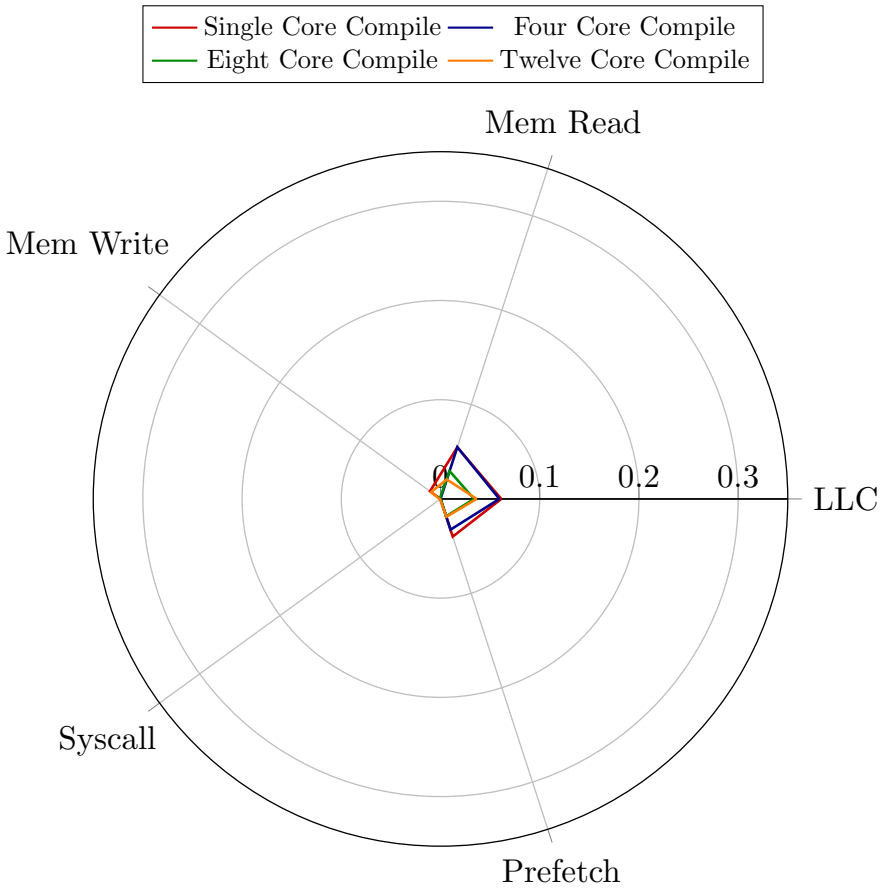


Figure 7.20: IPC Efficiency (IPCE) for compiling the Linux kernel with different amounts of cores.

7.3.5 Neighbor Microbenchmark

Figure 7.21 shows the sensitivity results of the neighbor microbenchmark when executed with different page sizes and in differently sized RDT partitions. The

reduced probing environment of six ways and 50% memory bandwidth introduces a bias. When using eleven ways, both containers (2 MB pages & 1 GB pages) can exploit the cache, hence their baseline IPCE is the same. If we reduce the allocation, the 2 MB page container is already starving during baseline establishment, so the aggressor's impact is smaller than on the 1 GB page container. This shows that sensitivity depends on all aspects, including the application, configuration, and host configuration.

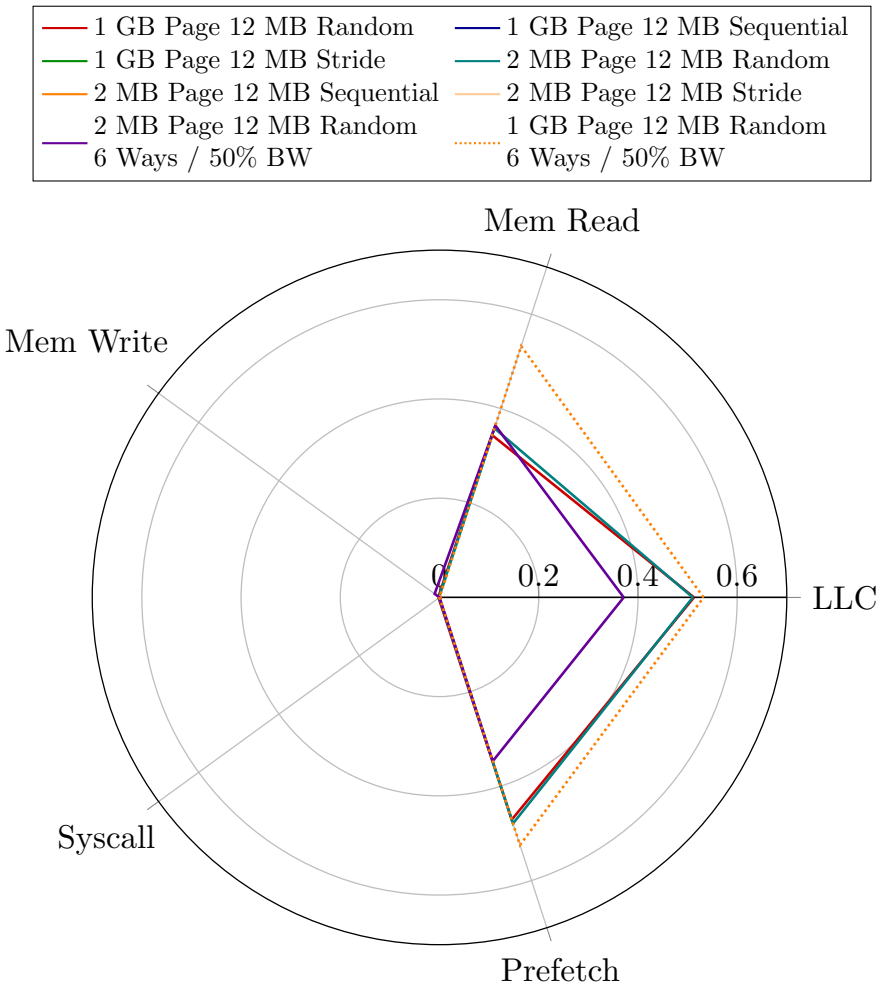


Figure 7.21: The IPC Efficiency sensitivity of the Neighbor microbenchmark with different page sizes.

7.3.6 Insensitive Implementations

At least in their default configurations, many microbenchmarks of stress-ng are purely CPU-bound, as shown in Figure 7.22.

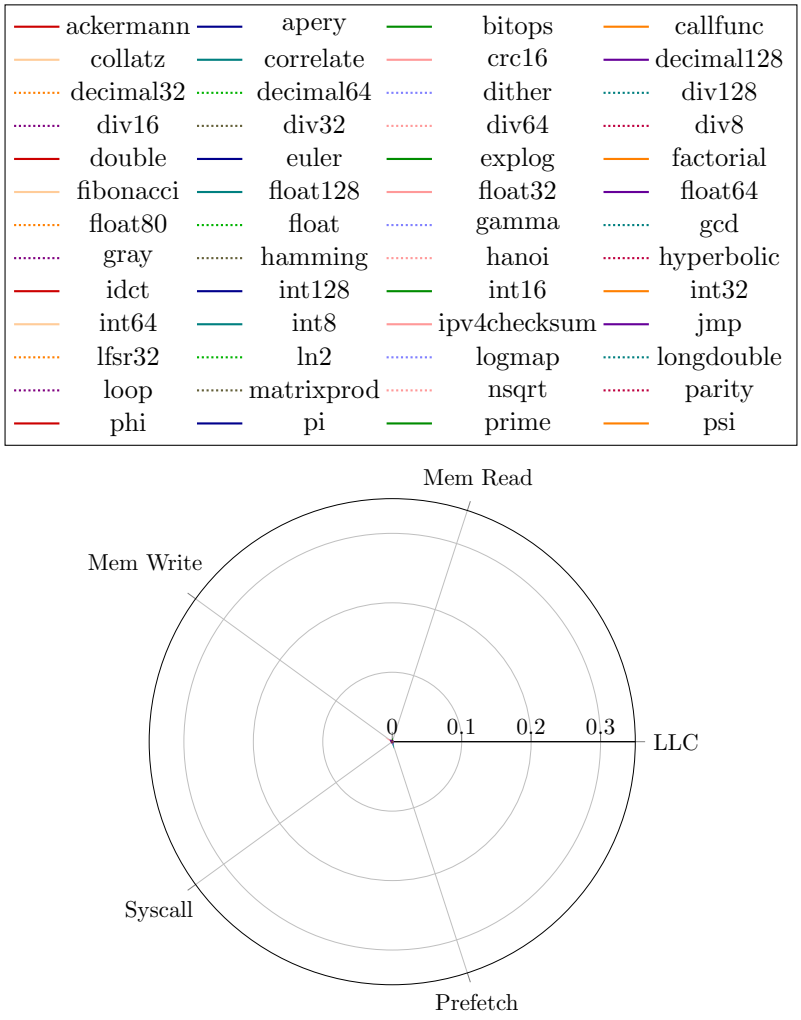


Figure 7.22: The IPC Efficiency of various implementations and methods which are insensitive. The microbenchmarks are used with their default configuration.

Chapter 8

Sustainability Analysis and Ethical Implications

As demand for computational resources continues to grow, the impact of data centers on global sustainability is becoming increasingly important. At the data center level, there are numerous aspects that define efficiency and sustainability [58, 59, 60]. Within the scope of this project, sustainability-aware task scheduling is, for example, addressed by Beena et al. [61]. The researchers developed carbon-aware scheduling for high-energy workloads. Nowadays, these typically include artificial intelligence training and inference jobs. The researchers derive carbon-intensity data as a measure of how "dirty" a workload is, based on the local energy grid at that moment. The system uses both real-time and historical data to derive the decision-making. Regarding our project, we saw how different workloads react to interference and partitioning. In the case of priority jobs that can achieve higher efficiency with a specific amount of exclusive resources, the improvement typically stems from reducing or eliminating practically all cache misses. From an energy perspective, each cache miss incurs additional energy because it requires accessing DRAM. In a large-scale system, this difference in efficiency could have practical implications for energy consumption and cooling demand. In combination with a carbon-aware scheduling approach, a job that was previously considered dirty could be turned into a cleaner job by using isolation. In return, this can give the respective schedulers a wider range of decisions. At this time, we do not see substantial ethical implications for this project.

Chapter 9

Conclusion

In this work, we successfully mitigated the impact of the noisy neighbor problem by employing isolation of shared resources via the Intel Resource Director Technology (RDT). To this end, we defined robust metrics to quantify the relation between resource availability and performance. We introduced a dynamic scheduling policy that uses an online probing method to mitigate the noisy neighbor problem. We implemented this policy and probing process at the container level and experimentally demonstrated that it improves global efficiency by up to 4.9% compared to a common least-loaded policy, while imposing only a small penalty of 0.9% efficiency on batch workloads. In terms of quality of service, we show that the dynamic scheduling policy can increase the efficiency of priority jobs by 42% at a penalty of 0.4% to batch jobs. Given a target-QoS threshold of 90% efficiency, the fraction of time spent above it can increase from 0.1% to 62.3%, and can be further increased by relaxing resource utilization constraints.

We demonstrated the patterns resulting from allocation combinations and different workloads. The analysis of these patterns is the foundation for enabling noisy neighbor mitigations beyond quality of service guarantees. Specifically, distinguishing the resource-to-performance relationship is key to achieving improvements. This includes workloads that benefit incrementally from more resources, while others perform significantly better once allocated a specific amount or combination of resources. Typically, this combination corresponds to the point where a workload's working set can reside in its last-level cache partition.

To back these experiments, we developed a comprehensive experimental setup. It integrates and orchestrates all aspects of the project, enabling reliability and reproducible experiments. Most importantly, it integrates the RDT interface and exposes its functionality to scheduler implementations, which drastically reduces the effort required for future development. By operating in a highly concurrent manner, the experimental setup can run large-scale benchmarks at high sampling frequencies. Likewise, access to the collected data from a scheduler's perspective is facilitated by providing thread-safe interfaces. The

driver's memory footprint is generally low and can be further reduced by dynamically adapting the collected data. This ensures minimal impact on the ongoing experiment. Further, the remaining instrumentation interference can be isolated using RDT.

Additionally, we proposed and implemented an online sensitivity probing process. Employing RDT resource partitioning yields reproducible metrics in noisy environments without the need for a dedicated probing host. We conclude that the process has potential for a multi-node setup, where a global scheduler needs finer-grain information about the workloads' sensitivities.

We provided a detailed analysis of the current technical limitations of RDT cache partitioning related to cache slicing and memory page configurations. In particular, we experimentally showed how last-level cache partitioning amplifies conflict cache misses for processes backed by standard 4 KB or transparent 2 MB pages. We connect this effect to the hash function, which maps physical addresses to cache slices. Lastly, we consider the impact of higher node utilization on sustainability.

In terms of future prospects, the sensitivity probing can be further refined to create more specific interference and to support a multi-node setup. To this end, the experimental setup can be transformed into a client-server model. Since other vendors (AMD, ARM) support similar resource partitioning interfaces, the scope could be extended to heterogeneous clusters. Likewise, the dynamic scheduling policy and the respective allocation probing could benefit from a wider range of resources. Specifically, a pool of hosts with different allocatable unit sizes could help mitigate the trade-off between brittle performance and unused resources. Currently, the scheduler is given hints about workload priorities. Future implementations should use the available real-time monitoring data and implement heuristics or models to automatically determine whether it is worth allocating exclusive resources to a container.

Bibliography

[1] Intel®. *Intel® Resource Director Technology (Intel® RDT) Framework*. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html. Accessed: 23 March 2025. 2025.

[2] Eli Cortez et al. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms". In: *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*. Oct. 2017. URL: https://www.microsoft.com/en-us/research/publication/resource-central-understanding-predicting-workloads-improved-resource-management-large-cloud-platforms/.

[3] Xiaoting Qin et al. "How Different are the Cloud Workloads? Characterizing Large-Scale Private and Public Cloud Workloads". In: June 2023, pp. 522–530. DOI: 10.1109/DSN58367.2023.00055.

[4] Hussain AlJahdali et al. "Multi-tenancy in cloud computing". In: *2014 IEEE 8th international symposium on service oriented system engineering*. IEEE. 2014, pp. 344–351.

[5] Daniel Olusegun Olabanji, Olumuyiwa Oluwafunto Matthew, and Tineke Fitch. "Multi-tenancy in cloud-native architecture: a systematic mapping study". In: *WSEAS Transactions on Computers* 22.4 (2023), pp. 25–43.

[6] Fangfei Liu et al. "Last-level cache side-channel attacks are practical". In: *2015 IEEE symposium on security and privacy*. IEEE. 2015, pp. 605–622.

[7] Amarnath Jasti et al. "Security in multi-tenancy cloud". In: *44th Annual 2010 IEEE International Carnahan Conference on Security Technology*. IEEE. 2010, pp. 35–41.

[8] George Kousiouris, Tommaso Cucinotta, and Theodora Varvarigou. "The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks". In: *Journal of Systems and Software* 84.8 (2011), pp. 1270–1291.

[9] Ruiqing Chi, Zhuzhong Qian, and Sanglu Lu. "Be a good neighbour: Characterizing performance interference of virtual machines under xen virtualization environments". In: *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2014, pp. 257–264.

[10] Christina Delimitrou and Christos Kozyrakis. "ibench: Quantifying interference for datacenter applications". In: *2013 IEEE international symposium on workload characterization (IISWC)*. IEEE. 2013, pp. 23–33.

[11] Eiman Ebrahimi et al. "Prefetch-aware shared resource management for multi-core systems". In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), pp. 141–152. ISSN: 0163-5964. DOI: 10.1145/2024723.2000081. URL: https://doi.org/10.1145/2024723.2000081.

[12] Alexander Zuepke and Robert Kaiser. "Deterministic Futexes: Addressing WCET and Bounded Interference Concerns". In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019, pp. 65–76. DOI: 10.1109/RTAS.2019.00014.

[13] Intel. *intel-cmt-cat*. https://github.com/intel/intel-cmt-cat/tree/master. Accessed: 1 January 2026. 2026.

[14] Intel. *Intel Xeon Silver 4314, Specifications*. https://www.intel.com/content/www/us/en/products/sku/215269/intel-xeon-silver-4314-processor-24m-cache-2-40-ghz/specifications.html. Accessed: 10 September 2025. 2025.

[15] Intel. *Intel Xeon Platinum 8360Y, Specifications*. https://www.intel.com/content/www/us/en/products/sku/212459/intel-xeon-platinum-8360y-processor-54m-cache-2-40-ghz/specifications.html. Accessed: 10 September 2025. 2025.

[16] Intel. *Intel Xeon Gold 6326, Specifications*. https://www.intel.com/content/www/us/en/products/sku/215274/intel-xeon-gold-6326-processor-24m-cache-2-90-ghz/specifications.html. Accessed: 10 September 2025. 2025.

[17] Intel. *Intel Xeon Platinum 8480, Specifications*. https://www.intel.com/content/www/us/en/products/sku/231746/intel-xeon-platinum-8480-processor-105m-cache-2-00-ghz/specifications.html. Accessed: 10 September 2025. 2025.

[18] Intel. *Intel Xeon Gold 5512U, Specifications*. https://www.intel.com/content/www/us/en/products/sku/237565/intel-xeon-gold-5512u-processor-52-5m-cache-2-10-ghz/specifications.html. Accessed: 10 September 2025. 2025.

[19] Intel. *Intel Xeon® Gold 6526Y, Specifications*. https://www.intel.com/content/www/us/en/products/sku/237560/intel-xeon-gold-6526y-processor-37-5m-cache-2-80-ghz/specifications.html. Accessed: 13 October 2025. 2025.

[20] Intel. *Intel® Xeon® Gold 6548Y+, Specifications*. https://www.intel.com/content/www/us/en/products/sku/237564/intel-xeon-gold-6548y-processor-60m-cache-2-50-ghz/specifications.html. Accessed: 13 October 2025. 2025.

[21] Intel. *Intel® Xeon® Gold 6548N, Specifications*. https://www.intel.com/content/www/us/en/products/sku/237567/intel-xeon-gold-6548n-processor-60m-cache-2-80-ghz/specifications.html. Accessed: 13 October 2025. 2025.

[22] Intel. *Intel® Xeon® CPU Max 9462, Specifications*. https://www.intel.com/content/www/us/en/products/sku/232597/intel-xeon-cpu-max-9462-processor-75m-cache-2-70-ghz/specifications.html. Accessed: 13 October 2025. 2025.

[23] Barcelona Supercomputing Center. *MareNostrum 5 Technical Information*. Accessed: 3 January 2026. Plaça Eusebi Güell, 1-3, 08034 Barcelona, 2026. URL: https://www.bsc.es/marenostrum/marenostrum-5.

[24] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. "Addressing shared resource contention in multicore processors via scheduling". In: *SIGARCH Comput. Archit. News* 38.1 (Mar. 2010), pp. 129–142. ISSN: 0163-5964. DOI: 10.1145/1735970.1736036. URL: https://doi.org/10.1145/1735970.1736036.

[25] Jason Mars et al. "Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. Porto Alegre, Brazil: Association for Computing Machinery, 2011, pp. 248–259. ISBN: 9781450310536. DOI: 10.1145/2155620.2155650. URL: https://doi.org/10.1145/2155620.2155650.

[26] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-aware scheduling for heterogeneous datacenters". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 77–88. ISBN: 9781450318709. DOI: 10.1145/2451116.2451125. URL: https://doi.org/10.1145/2451116.2451125.

[27] Jianyong Zhu et al. "Perph: A Workload Co-location Agent with Online Performance Prediction and Resource Inference". In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021, pp. 176–185. DOI: 10.1109/CCGrid51090.2021.00027.

[28] Apache Software Foundation. *Apache Hadoop YARN*. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html. Accessed: 3 January 2026. 2025.

[29] Docker Inc. *Docker Engine API Reference*. Accessed: 25 December 2025. Dec. 2025. URL: https://docs.docker.com/reference/api/engine/.

[30] Docker Inc. *Docker Client Go Package*. https://pkg.go.dev/github.com/docker/docker/client. Published: 5 November 2025, Accessed: 25 December 2025. Nov. 2025.

[31] Intel. *goresctrl Library*. https://github.com/intel/goresctrl. Accessed: 1 January 2026. 2026.

[32] Open Container Initiative. *runc*. https://github.com/opencontainers/runc/tree/main. Accessed: 1 January 2025.

[33] Intel Corporation. *Intel® Xeon® Processor Scalable Family: Specification Update (Errata SKX99)*. https://www.intel.com/content/www/us/en/content-details/613537/intel-xeon-processor-scalable-family-specification-update.html. Reference Number: 613537-033US. Accessed: 22 November 2025. Apr. 2023.

[34] Linux Foundation. *perf: Linux profiling with performance counters*. https://perfwiki.github.io/main/. Accessed: 1 November 2025.

[35] Elasticsearch B.V. *go-perf: perf for Go*. https://github.com/elastic/go-perf. Accessed: 1 November 2025.

[36] Linux Foundation. *Top-down analysis with the perf tool*. https://perfwiki.github.io/main/top-down-analysis/. Accessed: 1 November 2025.

[37] Colin Ian King. *stress-ng repository*. https://github.com/ColinIanKing/stress-ng. Accessed: 26 November 2025.

[38] Michael Kerrisk. *futex(2) Linux manual page*. https://man7.org/linux/man-pages/man2/futex.2.html. Accessed: 22 November 2025. 2025.

[39] openEuler Wangshuo. *Linux Futex Principle Analysis*. https://www.openeuler.org/en/blog/wangshuo/Linux_Futex_Principle_Analysis/. Accessed: 22 November 2025.

[40] Grafana Labs. *Grafana Homepage*. Accessed: 3 January 2026. 2026. URL: https://grafana.com/.

[41] The PostgreSQL Global Development Group. *PostgreSQL*. https://www.postgresql.org/. Accessed: 15 January 2026. 2026.

[42] Free Software Foundation, Inc. *GCC Compiler*. https://gcc.gnu.org/. Accessed: 15 January 2026. 2026.

[43] Igor Pavlov. *7-Zip*. https://www.7-zip.org/. Accessed: 15 January 2026. 2025.

[44] Fabrice Bellard. *FFmpeg*. https://www.ffmpeg.org/. Accessed: 15 January 2026. 2026.

[45] XGBoost Developers. *XGBoost Documentation*. https://xgboost.readthedocs.io/en/stable. Accessed: 15 January 2026. 2025.

[46] Barcelona Supercomputing Center. *MareNostrum 5 Technical Information*. https://www.bsc.es/marenostrum/marenostrum-5. Accessed: 14 September 2025. 2025.

[47] CloudLab. *CloudLab Machine Type Specification*. https://docs.cloudlab.us/hardware.html. Accessed: 12th September 2025. 2025.

[48] CloudLab Wisconsin. *CloudLab sm220u Machine, lshw*. https://www.wisc.cloudlab.us/portal/show-hardware.php?type=sm220u. Accessed: 13 October 2025. 2025.

[49] CloudLab Clemson. *CloudLab r650 Machine, lshw*. https://www.clemson.cloudlab.us/portal/show-hardware.php?type=r650. Accessed: 13 October 2025. 2025.

[50] CloudLab Utah. *CloudLab c6620 Machine, lshw*. https://www.utah.cloudlab.us/portal/show-hardware.php?type=c6620. Accessed: 12 September 2025. 2025.

[51] Parul Sohal et al. "A Closer Look at Intel Resource Director Technology (RDT)". In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. RTNS '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 127–139. ISBN: 9781450396509. DOI: 10.1145/3534879.3534882. URL: https://doi.org/10.1145/3534879.3534882.

[52] Alireza Farshin et al. "Make the Most out of Last Level Cache in Intel Processors". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303977. URL: https://doi.org/10.1145/3302424.3303977.

[53] John D. McCalpin. *Intel Address Hash Tool*. https://github.com/jdmccalpin/Intel_Address_Hash. GitHub repository, Accessed: 25 December 2025. 2025.

[54] Fangfei Liu et al. "Last-Level Cache Side-Channel Attacks are Practical". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP '15. USA: IEEE Computer Society, 2015, pp. 605–622. ISBN: 9781467369497. DOI: 10.1109/SP.2015.43. URL: https://doi.org/10.1109/SP.2015.43.

[55] John D. McCalpin. *Mapping Addresses to L3/CHA Slices in Intel Processors*. Tech. rep. TR-2021-03. Texas Advanced Computing Center (TACC), Sept. 2021. DOI: 10.26153/tsw/14539. URL: https://hdl.handle.net/2152/87595.

[56] Intel. *k8s-rdt-controller Repository*. https://github.com/intel/k8s-rdt-controller. Accessed: January 17, 2026. 2026.

[57] 7zip Open Source Doumentation. *Set Sompression Method Options*. https://7-zip.opensource.jp/chm/cmdline/switches/method.htm. Accessed: 16 January 2025. 2025.

[58] Jordi Guitart. "Toward sustainable data centers: a comprehensive energy management strategy". In: *Computing* 99.6 (June 2017), pp. 597–615. ISSN: 0010-485X. DOI: 10.1007/s00607-016-0501-1. URL: https://doi.org/10.1007/s00607-016-0501-1.

[59] Daniel Raphael Ejike Ewim et al. “Impact of data centers on climate change: a review of energy efficient strategies”. In: *The Journal of Engineering and Exact Sciences* 9.6 (2023), 16397–01e.

[60] Zhiwei Cao et al. “Data Center Sustainability: Revisits and Outlooks”. In: *IEEE Transactions on Sustainable Computing* 9.3 (2024), pp. 236–248. DOI: 10.1109/TSUSC.2023.3281583.

[61] B. M. Beena et al. “A Green Cloud-Based Framework for Energy-Efficient Task Scheduling Using Carbon Intensity Data for Heterogeneous Cloud Servers”. In: *IEEE Access* 13 (2025), pp. 73916–73938. DOI: 10.1109/ACCESS.2025.3562882.

Appendix A

Implementation Details

A.1 Collection Data

```
type PerfMetrics struct {
    CacheMisses      *uint64
    CacheReferences  *uint64
    Instructions     *uint64
    Cycles           *uint64
    BranchInstructions *uint64
    BranchMisses     *uint64
    BusCycles        *uint64

    StallsTotal            *uint64
    StallsL3Miss           *uint64
    StallsL2Miss           *uint64
    StallsL1dMiss          *uint64
    StallsMemAny           *uint64
    ResourceStallsSB       *uint64
    ResourceStallsScoreboard *uint64

    L1DCacheLoadMisses *uint64
    L1DCacheLoads    *uint64
    L1DCacheStores   *uint64
    L1ICacheLoadMisses *uint64
    LLCLoadMisses    *uint64
    LLCLoads         *uint64
    LLCStoreMisses   *uint64
    LLCStores        *uint64

    // Derived metrics
    CacheMissRate      *float64
    InstructionsPerCycle *float64
    StalledCyclesPercent *float64
    StallsL3MissPercent *float64
```

```
    TheoreticalIPC     *float64
    IPCEfficiency      *float64
}
```

Listing A.1: The `PerfMetrics` structure.

```
type RDTMetrics struct {
    RDTClassName *string
    MonGroupName *string
    MBAThrottle *uint64

    // Per-socket monitoring metrics
    L3OccupancyPerSocket       map[int]uint64
    MemoryBandwidthTotalPerSocket map[int]uint64
    MemoryBandwidthLocalPerSocket map[int]uint64
    L3UtilizationPctPerSocket  map[int]float64
    MemBandwidthMbpsPerSocket  map[int]float64

    // Per-socket allocation details
    L3BitmaskPerSocket     map[int]string
    L3WaysPerSocket        map[int]uint64
    L3AllocationPctPerSocket map[int]float64
    MBAPercentPerSocket    map[int]uint64

    // Full allocation strings from resctrl schemata
    L3AllocationString *string
    MBAAllocationString *string
}
```

Listing A.2: The `RDTMetrics` structure.

```
type DockerMetrics struct {
    CPUUsageTotal    *uint64
    CPUUsageKernel   *uint64
    CPUUsageUser     *uint64
    CPUUsagePercent  *float64
    CPUThrottling    *uint64
    MemoryUsage      *uint64
    MemoryLimit      *uint64
    MemoryCache      *uint64
    MemoryRSS        *uint64
    MemorySwap       *uint64
    MemoryUsagePercent *float64
    NetworkRxBytes   *uint64
    NetworkTxBytes   *uint64
    DiskReadBytes    *uint64
    DiskWriteBytes   *uint64

    // Scheduler-assigned CPU affinity metadata
    AssignedCores  map[int]bool
```

```
    AssignedCoresCSV *string
}
```

Listing A.3: The `DockerMetrics` structure.

A.2 A findBestFitBitmaskForWays Implementation

```
func (a *RDTAccountant) findBestFitBitmaskForWays(targetWays int, state
    *SocketState) (string, error) {
    if targetWays <= 0 {
        return "", fmt.Errorf("invalid target ways: %d (must be > 0)",
            targetWays)
    }
    if targetWays > state.TotalWays {
        return "", fmt.Errorf("requested %d ways but only %d available",
            targetWays, state.TotalWays)
    }

    // Find contiguous unallocated bits
    allocated := state.AllocatedBitmask
    bestStart := -1
    bestLength := 0

    for start := 0; start < state.TotalWays; start++ {
        if (allocated & (1 << start)) != 0 {
            continue // This way is allocated
        }

        // Count contiguous free ways starting from here
        length := 0
        for i := start; i < state.TotalWays && (allocated&(1<<i)) == 0;
            i++ {
            length++
        }

        // Check if this is a better fit (prefer smallest sufficient
            region)
        if length >= targetWays && (bestStart == -1 || length <
            bestLength) {
            bestStart = start
            bestLength = length
        }
    }

    if bestStart == -1 {
        return "", fmt.Errorf("no contiguous %d cache ways available",
            targetWays)
```

```
    }

    // Create bitmask for targetWays starting at bestStart
    mask := uint64(0)
    for i := 0; i < targetWays; i++ {
        mask |= (1 << (bestStart + i))
    }

    return fmt.Sprintf("0x%x", mask), nil
}
```

Listing A.4: A `findBestFitBitmaskForWays` implementation.

A.3 The applyManagedConfigLocked Function

```
func (a *DefaultRDTAllocator) applyManagedConfigLocked(force bool) error {
    a.logger.WithField("total_classes", len(a.managedConfigs)).Debug("Creating all RDT classes in single configuration")

    configClasses := make(map[string]struct {
        L2Allocation rdt.CatConfig      `json:"l2Allocation"`
        L3Allocation rdt.CatConfig      `json:"l3Allocation"`
        MBAllocation rdt.MbaConfig      `json:"mbAllocation"`
        Kubernetes  rdt.KubernetesOptions `json:"kubernetes"`
    }, len(a.managedConfigs))

    for className, classConfig := range a.managedConfigs {
        l3Config := rdt.CatConfig{}
        if classConfig.Socket0 != nil {
            l3Alloc, err := a.resolveL3Allocation(classConfig.Socket0)
            if err != nil {
                return fmt.Errorf("failed to resolve L3 allocation for %s socket0: %w", className, err)
            }
            if l3Alloc != "" {
                l3Config["0"] = rdt.CacheIdCatConfig{Unified: l3Alloc}
            }
        }
        if classConfig.Socket1 != nil {
            l3Alloc, err := a.resolveL3Allocation(classConfig.Socket1)
            if err != nil {
                return fmt.Errorf("failed to resolve L3 allocation for %s socket1: %w", className, err)
            }
            if l3Alloc != "" {
                l3Config["1"] = rdt.CacheIdCatConfig{Unified: l3Alloc}
```

```
        }
    }

    mbConfig := rdt.MbaConfig{}
    if classConfig.Socket0 != nil {
        if mbAlloc := resolveMBAllocation(classConfig.Socket0); mbAlloc != "" {
            mbConfig["0"] = rdt.CacheIdMbaConfig{mbAlloc}
        }
    }
    if classConfig.Socket1 != nil {
        if mbAlloc := resolveMBAllocation(classConfig.Socket1); mbAlloc != "" {
            mbConfig["1"] = rdt.CacheIdMbaConfig{mbAlloc}
        }
    }

    configClasses[className] = struct {
        L2Allocation rdt.CatConfig      `json:"l2Allocation"`
        L3Allocation rdt.CatConfig      `json:"l3Allocation"`
        MBAllocation rdt.MbaConfig      `json:"mbAllocation"`
        Kubernetes  rdt.KubernetesOptions `json:"kubernetes"`
    }{
        L3Allocation: l3Config,
        MBAllocation: mbConfig,
    }
}

config := &rdt.Config{
    Partitions: map[string]struct {
        L2Allocation rdt.CatConfig `json:"l2Allocation"`
        L3Allocation rdt.CatConfig `json:"l3Allocation"`
        MBAllocation rdt.MbaConfig `json:"mbAllocation"`
        Classes     map[string]struct {
            L2Allocation rdt.CatConfig      `json:"l2Allocation"`
            L3Allocation rdt.CatConfig      `json:"l3Allocation"`
            MBAllocation rdt.MbaConfig      `json:"mbAllocation"`
            Kubernetes  rdt.KubernetesOptions `json:"kubernetes"`
        } `json:"classes"`
    }{
        "": {
            L3Allocation: rdt.CatConfig{
                rdt.CacheIdAll: rdt.CacheIdCatConfig{Unified: rdt.CacheProportion("100%")},
            },
            MBAllocation: rdt.MbaConfig{
                rdt.CacheIdAll: rdt.CacheIdMbaConfig{rdt.MbProportion("100%")},
            },
            Classes: configClasses,
```

```
            },
        },
    }

    rdtguard.Lock()
    err := rdt.SetConfig(config, force)
    rdtguard.Unlock()
    if err != nil {
        return fmt.Errorf("failed to create RDT classes: %v", err)
    }

    a.logger.WithField("classes_created",
        len(a.managedConfigs)).Debug("All RDT classes created
        successfully")
    return nil
}
```

Listing A.5: The `applyManagedConfigLocked` implementation.

A.4 FFmpeg Benchmarking Commands

```
# Blur
ffmpeg -i /data/video.mov -vf 'boxblur=10:1' -c:v libx264 -preset fast
    -crf 23 -c:a copy -y /tmp/output.mp4

# Filter
ffmpeg -i /data/video.mov -vf 'scale=640:360,hue=s=0' -c:v libx264
    -preset medium -crf 23 -c:a copy -y /tmp/output.mp4

# High Compression
ffmpeg -i /data/video.mov -c:v libx265 -preset veryslow -crf 22 -c:a aac
    -b:a 128k -y /tmp/output.mp4
```

Listing A.6: The used FFmpeg benchmarking commands.

A.5 The Neighbor Microbenchmark

```c
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include <errno.h>
#include <stdint.h>
#include <limits.h>
#include <sys/mman.h>

static volatile int running = 1;

typedef enum {
    ALLOC_MALLOC = 0,
    ALLOC_THP,
    ALLOC_HUGETLB_2M,
    ALLOC_HUGETLB_1G,
} alloc_mode_t;

typedef struct {
    char *ptr;
    long size;
    alloc_mode_t mode;
} buffer_alloc_t;

...

#ifndef MAP_HUGETLB
#define MAP_HUGETLB 0x40000
#endif

#ifndef MAP_HUGE_SHIFT
#define MAP_HUGE_SHIFT 26
#endif

#ifndef MAP_HUGE_2MB
#define MAP_HUGE_2MB (21 << MAP_HUGE_SHIFT)
#endif

#ifndef MAP_HUGE_1GB
#define MAP_HUGE_1GB (30 << MAP_HUGE_SHIFT)
#endif

static const char *alloc_mode_to_string(alloc_mode_t mode) {
```

```
    switch (mode) {
        case ALLOC_MALLOC: return "none";
        case ALLOC_THP: return "thp";
        case ALLOC_HUGETLB_2M: return "hugetlb-2m";
        case ALLOC_HUGETLB_1G: return "hugetlb-1g";
        default: return "unknown";
    }
}

static int parse_alloc_mode(const char *mode_str, alloc_mode_t
    *out_mode) {
    if (mode_str == NULL || out_mode == NULL) {
        return -1;
    }
    if (strcmp(mode_str, "none") == 0) {
        *out_mode = ALLOC_MALLOC;
        return 0;
    }
    if (strcmp(mode_str, "thp") == 0) {
        *out_mode = ALLOC_THP;
        return 0;
    }
    if (strcmp(mode_str, "hugetlb-2m") == 0) {
        *out_mode = ALLOC_HUGETLB_2M;
        return 0;
    }
    if (strcmp(mode_str, "hugetlb-1g") == 0) {
        *out_mode = ALLOC_HUGETLB_1G;
        return 0;
    }
    return -1;
}

static int alloc_buffer(long buffer_size, alloc_mode_t mode,
    buffer_alloc_t *out_alloc) {
    if (out_alloc == NULL || buffer_size <= 0) {
        errno = EINVAL;
        return -1;
    }

    out_alloc->ptr = NULL;
    out_alloc->size = buffer_size;
    out_alloc->mode = mode;

    if (mode == ALLOC_MALLOC) {
        out_alloc->ptr = (char *)malloc((size_t)buffer_size);
        return out_alloc->ptr ? 0 : -1;
    }

    long map_size = buffer_size;
```

```
    int flags = MAP_PRIVATE | MAP_ANONYMOUS;
    if (mode == ALLOC_HUGETLB_2M || mode == ALLOC_HUGETLB_1G) {
        flags |= MAP_HUGETLB;
        if (mode == ALLOC_HUGETLB_2M) {
            flags |= MAP_HUGE_2MB;
            map_size = round_up_multiple(buffer_size, 2L * 1024 * 1024);
        } else {
            flags |= MAP_HUGE_1GB;
            map_size = round_up_multiple(buffer_size, 1024L * 1024 *
                1024);
        }
    }

    if (map_size <= 0) {
        if (errno == 0) {
            errno = EINVAL;
        }
        return -1;
    }

    void *p = mmap(NULL, (size_t)map_size, PROT_READ | PROT_WRITE,
        flags, -1, 0);
    if (p == MAP_FAILED) {
        out_alloc->ptr = NULL;
        return -1;
    }
    out_alloc->ptr = (char *)p;
    out_alloc->size = map_size;

    if (mode == ALLOC_THP) {
        // Best-effort: encourages transparent huge pages (usually 2MB;
            1GB THP depends on kernel settings).
#ifdef MADV_HUGEPAGE
        (void)madvise(out_alloc->ptr, (size_t)map_size, MADV_HUGEPAGE);
#endif
    }
    return 0;
}

...

void random_access(char *buffer, long size, int duration) {
    time_t start_time = time(NULL);
    srand(time(NULL));
    long operations = 0;

    printf("Starting random access pattern for %d seconds with buffer
        size %ld bytes\n", duration, size);

    while (running && (time(NULL) - start_time) < duration) {
```

```
        // Access multiple random locations per iteration for intensive memory pressure
        for (int burst = 0; burst < 10000 && running; burst++) {
            long offset = rand() % size;
            volatile char temp = buffer[offset];
            buffer[offset] = (char)((temp + offset) % 256);
            operations++;
        }
    }

    printf("Random access completed: %ld operations\n", operations);
}

void stride_access(char *buffer, long size, int duration) {
    time_t start_time = time(NULL);
    long stride = 4096; // 4KB stride for cache-unfriendly access
    long pos = 0;
    long operations = 0;

    printf("Starting stride access pattern for %d seconds with buffer size %ld bytes\n", duration, size);

    while (running && (time(NULL) - start_time) < duration) {
        // Access multiple stride locations per iteration
        for (int burst = 0; burst < 1000 && running; burst++) {
            volatile char temp = buffer[pos];
            buffer[pos] = (char)((temp + pos) % 256);
            pos = (pos + stride) % size;
            operations++;
        }
    }

    printf("Stride access completed: %ld operations\n", operations);
}

void sequential_access(char *buffer, long size, int duration) {
    time_t start_time = time(NULL);
    long pos = 0;
    long operations = 0;
    long total_passes = 0;

    printf("Starting sequential access pattern for %d seconds with buffer size %ld bytes\n", duration, size);

    while (running && (time(NULL) - start_time) < duration) {
        for (long i = 0; i < size && running && (time(NULL) - start_time) < duration; i += sizeof(long)) {
            volatile long *ptr = (long*)(buffer + i);
            *ptr = (*ptr + i) % 0xFFFFFFFF;
            operations++;
```

```
        }
        total_passes++;
        printf("Sequential pass %ld completed\n", total_passes);
    }

    printf("Sequential access completed: %ld operations, %ld passes\n",
        operations, total_passes);
}

int main(int argc, char *argv[]) {
    int duration = 3600;
    long buffer_size = 100 * 1024 * 1024;
    char pattern[32] = "random";
    alloc_mode_t alloc_mode = ALLOC_MALLOC;

    signal(SIGTERM, signal_handler);
    signal(SIGINT, signal_handler);

    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "--duration") == 0) {
            if (i + 1 < argc) {
                duration = atoi(argv[++i]);
                if (duration <= 0) {
                    fprintf(stderr, "Invalid duration: %s\n", argv[i]);
                    return 1;
                }
            } else {
                fprintf(stderr, "--duration requires a value\n");
                return 1;
            }
        } else if (strcmp(argv[i], "--buffer-size") == 0) {
            if (i + 1 < argc) {
                buffer_size = parse_size(argv[++i]);
                if (buffer_size <= 0) {
                    fprintf(stderr, "Invalid buffer size: %s\n", argv[i]);
                    return 1;
                }
            } else {
                fprintf(stderr, "--buffer-size requires a value\n");
                return 1;
            }
        } else if (strcmp(argv[i], "--pattern") == 0) {
            if (i + 1 < argc) {
                strncpy(pattern, argv[++i], sizeof(pattern) - 1);
                pattern[sizeof(pattern) - 1] = '\0';
                if (strcmp(pattern, "random") != 0 &&
                    strcmp(pattern, "stride") != 0 &&
                    strcmp(pattern, "sequential") != 0) {
                    fprintf(stderr, "Invalid pattern: %s. Use random,
                        stride, or sequential\n", pattern);
```

```
                    return 1;
                }
            } else {
                fprintf(stderr, "--pattern requires a value\n");
                return 1;
            }
        } else if (strcmp(argv[i], "--help") == 0) {
            print_usage(argv[0]);
            return 0;
        } else if (strcmp(argv[i], "--hugepages") == 0) {
            if (i + 1 < argc) {
                if (parse_alloc_mode(argv[++i], &alloc_mode) != 0) {
                    fprintf(stderr, "Invalid --hugepages mode: %s. Use
                        none, thp, hugetlb-2m, or hugetlb-1g\n", argv[i]);
                    return 1;
                }
            } else {
                fprintf(stderr, "--hugepages requires a value\n");
                return 1;
            }
        } else {
            fprintf(stderr, "Unknown option: %s\n", argv[i]);
            print_usage(argv[0]);
            return 1;
        }
    }

...
    return 0;
}
```

Listing A.7: Parts of the code of the neighbor microbenchmark. The benchmark accesses a buffer in random, sequential, and stride access patterns. It features different explicit page size request.

Appendix B

Supplement Data

B.1 Caching Home Agent Mappings

```
sudo ./Map_Addresses_to_L3_Slices.exe
PAGE_ADDRESSES
0x002241600000 0x00223d600000 0x002246400000 0x002241800000
    0x002c01e00000 0x002c04c00000 0x002ae0800000 0x002c02c00000
0x002d28e00000 0x002c06400000 0x002c06600000 0x002c05000000
    0x002c05200000 0x002c07400000 0x002c07600000 0x002c07000000
...
0x0021df200000 0x0021d9800000 0x0021d9a00000 0x0021d9c00000
    0x0021d9e00000 0x0021df400000 0x0021df600000 0x002144800000
0x002144a00000 0x002140c00000 0x002140e00000 0x002140400000
    0x002140600000 0x002140800000 0x002140a00000
CPUID Signature 0x606a0 identified as Ice Lake Xeon
DEBUG: TSC on core 0 socket 0 is 2611032282706353
DEBUG: TSC on core 63 socket 1 is 2611032283024791
CPUID Signature 0x606a0 identified as Ice Lake Xeon
CPUID Signature 0x606a0 identified as Ice Lake Xeon
INFO: CHA_PERFEVTSEL[0] = 0x400350
INFO: CHA_PERFEVTSEL[1] = 0x400350
INFO: CHA_PERFEVTSEL[2] = 0x400350
INFO: CHA_PERFEVTSEL[3] = 0x400350
SUCCESS: wrote mapping file 0 PADDR_0x002241600000.map
...
SUCCESS: wrote mapping file 13 PADDR_0x002d1cc00000.map
SUCCESS: wrote mapping file 14 PADDR_0x002d5ce00000.map
SUCCESS: wrote mapping file 15 PADDR_0x002142c00000.map
INFO: 16 new 2MiB pages have been mapped
DUMMY: globalsum 524618000
VERBOSE: L3 Mapping Complete in 524618 tries for 524288 cache lines
    ratio 1.000629
------------
LINES_BY_CHA
```

```
0 32768
1 32768
2 32768
3 32768
4 32768
5 32768
6 32768
7 32768
8 32768
9 32768
10 32768
11 32768
12 32768
13 32768
14 32768
15 32768
16 0
17 0
18 0
19 0
20 0
21 0
22 0
23 0
24 0
25 0
26 0
27 0
28 0
29 0
30 0
31 0
32 0
33 0
34 0
35 0
36 0
37 0
38 0
39 0
ACCCOUNTED FOR 524288 lines expected 524288 lines
```

Listing B.1: The Caching Home Agent mappings, which were determined using John McCalpin respective tool [53]. The output was shortened.

B.2 FFmpeg Sensitivities

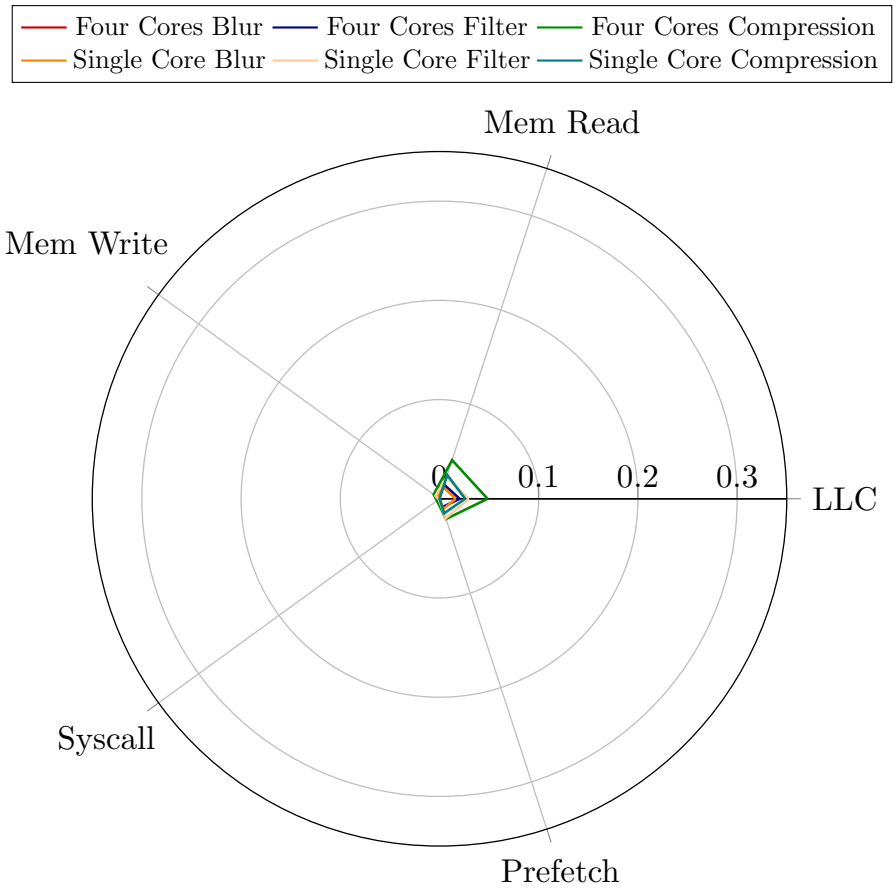


Figure B.1: The IPC Efficiency sensitivity of different FFmpeg methods.

B.3 Model Training Sensitivities

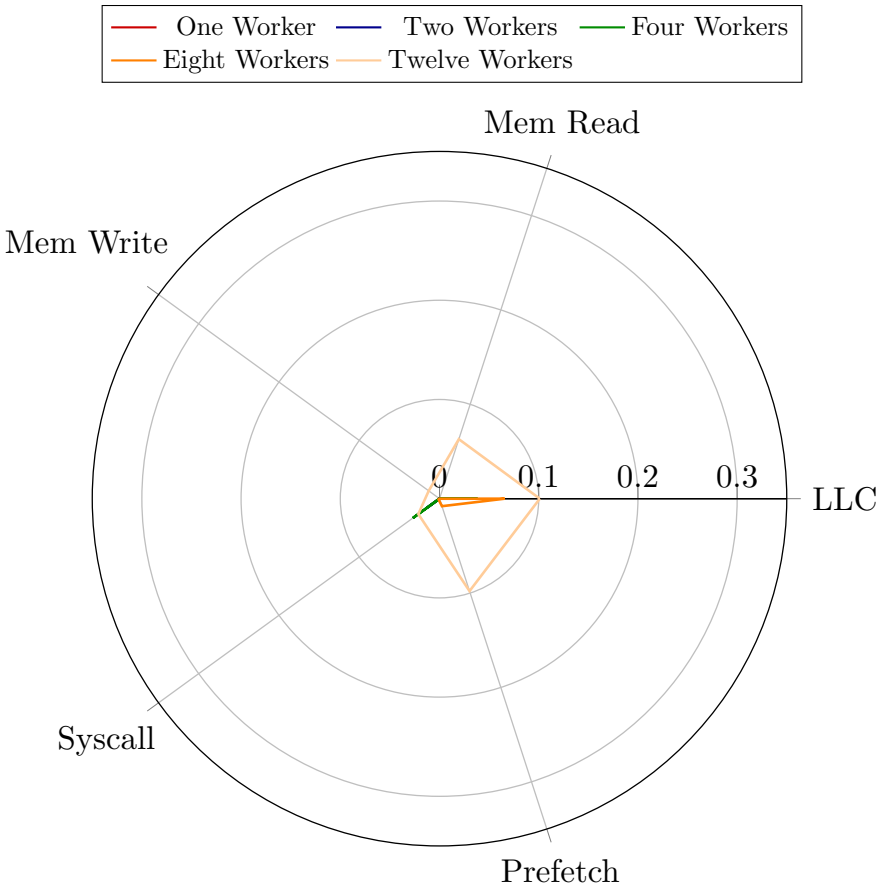


Figure B.2: IPC Efficiency (IPCE) of training a model with different amounts of worker processes.